DESARROLLO DE UNA HERRAMIENTA PARA DISCERNIR ERRORES MÚLTIPLES Y ERRORES SIMPLES EN MEMORIAS SRAM SOMETIDAS A RADIACIÓN

Illescas García, Paula
Zapico Suárez, David

GRADO EN INGENIERÍA DE COMPUTADORES. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería de Computadores

Madrid, 13 de Septiembre de 2017

Directores:

Mecha López, Hortensia
Clemente, Juan Antonio

Autorización de difusión

Illescas García, Paula
Zapico Suárez, David

Madrid, a 1 de septiembre de 2017

Los abajo firmantes, matriculados en el Grado de Ingeniería de Computadores de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "DESARROLLO DE UNA HERRAMIENTA PARA DISCERNIR ERRORES MÚLTIPLES Y ERRORES SIMPLES EN MEMORIAS SRAM SOMETIDAS A RADIACIÓN", realizado durante el curso académico 2016-2017 bajo la dirección de Hortensia Mecha López y la co-dirección de Juan Antonio Clemente en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Agradecimientos

A mis padres y hermano, gracias por todo

David

A mis padres y a mi familia, muchas gracias.

Paula

Índice general

Índice de figuras

Índice de tablas

Resumen

Para testear la vulnerabilidad de las memorias en entornos radiactivos se utilizan fuentes de radiación que en un breve lapso de tiempo, permiten obtener grandes cantidades de errores. Durante estos tests, se mezclan los errores simples (una partícula afecta a una celda de memoria) con los múltiples (una partícula afecta a múltiples celdas de memoria), y no es en absoluto trivial discernir ambos tipos de errores. Sin embargo, para mejorar la precisión de las métricas que evalúan la sensibilidad es imprescindible saber qué errores son simples y cuáles son múltiples.

En este trabajo, nos familiarizaremos con las distintas técnicas para distinguir los tipos de errores mencionados anteriormente. Después crearemos un software con las técnicas necesarias para la prediccion de dichos fallos usando el lenguaje C. Y finalmente desarrollamos una interfaz gráfica que será sencilla e intutitiva para facilitar la obtención de los datos y una mejora en la visualización.

Palabras clave

Multiple Cell Upset, Single Bit Upset, Single Events, Soft Errors, SRAMs

Abstract

To test the vulnerability of memories in radioactive environments, radiation sources are used to obtain large amounts of errors. During these tests, simple errors (a particle affects a memory cell) are mixed with the multiples (one particle affects multiple memory cells), and it is not at all trivial to discern both types of errors. However, to improve the accuracy of the metrics that evaluate the sensitivity it is imperative to know which errors are simple and which are multiple.

In this project, we will familiarize ourselves with the different techniques to tell apart the types of errors mentioned above. Then we will create a software with the necessary techniques for the prediction of such failures using the C language. And finally we developed a graphical interface that will be simple and intuitive to facilitate the obtaining of the data and an improvement in the visualization.

Keywords

Multiple Cell Upset, Single Bit Upset, Single Events, Soft Errors, SRAMs

Capítulo 1

Introducción

La evolución de la tecnología electrónica ha permitido que los transistores alcancen tamaños realmente diminutos. Como consecuencia, también ha crecido el riesgo de compartimiento de carga, así como la ocurrencia de eventos múltiples en *Memorias de Acceso Aleatorio Estáticas (Static Random Access Memories, SRAMs)*.

Los efectos que produce la radiación en este tipo de memorias no es un problema nuevo ya que desde hace varias décadas, se contempla a la hora de su fabricación y diseño para distintos ámbitos, como por ejemplo en la industria aeroespacial.

Es obligatorio establecer la distinción entre errores simples o SBUs *Single Bit Upsets* y errores múltiples o MCUs *Multiple Cell Upsets* al calcular las secciones eficaces para con radiación en SRAMs. Pero esto no es fácil de realizar. En las SRAMs antiguas, los bits de la misma palabra se ubicaban en localizaciones adyacentes, por lo que los errores múltiples se podían descubrir como varios trastornos de bits en la misma palabra *Multiple Bit Upsets, MBUs*. Teniendo en cuenta que estos errores no podían corregirse con las técnicas estándar como "Error Correcting Codes (ECC)", los fabricantes comenzaron a usar el "entrelazado" (interleaving) en las memorias. El objetivo era prevenir que estos eventos múltiples ocurrieran en la misma dirección y de este modo hacer que la recuperación frente a errores fuera más fácil.

Esto es un reto, porque a no ser que la estructura interna de la SRAM sea conocida, no es posible determinar la relación entre las direcciones lógicas y las físicas en las memorias. Desafortunadamente esta información los fabricantes la suelen mantener en privado y no está disponible. Por lo tanto, basándose en las anomalías estadísticas de los conjuntos de direcciones afectadas por radiación, se han desarrollado distintas estrategias para agrupar las direcciones en errores múltiples.

Podemos enmarcar este trabajo dentro de los estudios de la resistencia y sensibilidad a la radiación de los circuitos integrados, en particular de las memorias. Durante los últimos años, los fabricantes intentan satisfacer la demanda de mejora de prestaciones y una disminución de tamaño. Esto, junto con el escalado de la tecnología electrónica, puede que no garantice la estabilidad de las celdas y acarree un aumento de la susceptibilidad a dicha radiación, produciendo errores de funcionamiento. Existe un conocido problema que ocurre cuando las memorias se prueban bajo radiación: la determinación de si un trastorno en un bit es aislado, o si éste pertenece a un evento múltiple. Como es excepcional conocer el diseño físico de la memoria, este trabajo plantea el evaluar las propiedades estadísticas de un conjunto de direcciones con trastornos en sus bits y comparar los resultados con las predicciones matemáticas de un modelo donde todos los eventos son aislados. Así, un conjunto de reglas fáciles de implementar en lenguajes de programación comunes pueden ser aplicadas iterativamente si se observan anomalías. Esto nos permite una clasificación de los errores bastante cercana a la realidad con más del 80 % de exactitud.

En trabajos anteriores, se demostró la factibilidad de la detección de direcciones implicadas en eventos múltiples usando la operación XOR, que se propuso por primera vez en 2007[1]

Las direcciones que contenían bitflips se combinaban en pares, se aplicaba sobre ellas la operación XOR y finalmente, se seleccionaban los valores que aparecían más veces de lo esperado. El punto clave, era que la operación XOR obtiene ventaja de la modularidad de las SRAMs. Esto constituía una gran diferencia respecto a trabajos anteriores[2] que se usaron como punto de partida, donde a las direcciones se les aplicaba una operación de resta, en lugar de realizar sobre ellas la operación XOR. Sin embargo, es interesante encontrar un procedimiento general para separar la aleatoriedad de la causalidad.

1.1. Objetivo

A la hora de comenzar nuestro proyecto, el primer paso es definir los objetivos del trabajo, algo muy importante ya que nos marca una orientación hacia donde queremos llegar y avanzar en el proyecto. Los objetivos de este proyecto pueden dividirse en:

- Familiarizarnos con las distintas técnicas existentes para detectar los distintos tipos de errores.
- Desarrollar una herramienta software que implemente dichas técnicas en lenguaje C/C++.
- La creación de una interfaz gráfica sencilla y fácil de usar por el usuario.

El primer objetivo consiste en estudiar las distintas técnicas existentes para la detección de los distintos tipos de errores que se pueden dar en las memorias SRAM cuando son incididas por radiación. Este tipo de técnicas requiere del estudio de númerosos conceptos matemáticos básicos para la realización de las distintas funciones que calculan y previenen los errores. Hay que mencionar que las matemáticas juegan un rol especial en el pensamiento científico, proporcionando veracidad y rigor al estudio.

El segundo objetivo y podríamos decir el principal de este proyecto, es el desarrollo de un software para el análisis de los resultados obtenidos al irradiar memorias SRAM desde distintas fuentes de radiación, permitiendo discernir entre errores simples y errores multiples, y así caracterizar su sensibilidad frente a errores en entornos radiactivos, como por ejemplo, el espacio. Se utilizará el lenguaje C para su desarrollo ya que consideramos que reducirá su tiempo de computo, además de conseguir una mejor estructuracion del codigo. Hay que mencionar que las distintas técnicas para distinguir los tipos de errores están escritos en lenguaje Julia. Un lenguaje de programación de tipado dinámico de alto nivel y alto desempeño para la computación genérica, tecnica y científica, que requiere de su asimilación y aprendizaje previo con el fin de abstraer toda la información posible para implementar las tecnicas más relevantes para la realización del análisis.

Y finalmente como ultimo objetivo, y no por ello menos importante, fijamos crear una interfaz gráfica no muy compleja para poder trabajar de manera sencilla con ella, interactuar y visualizar los resultados obtenidos de los distintos experimentos.

1.2. Plan de trabajo

La finalidad de este proyecto es la de desarrollar un software para analizar los resultados que se obtengan al irradiar memorias SRAM a diferentes fuentes de radiación, y así caracterizar su sensibilidad frente a errores en entornos radiactivos como el espacio. Para ello nos familiarizarnos con las distintas técnicas diferentes para discernir dichos tipos de errores y posteriormente de desarrollaremos uns software en el lenguaje y plataforma que elijamos para implementar las técnicas mas relevantes, que sea intuitivo y que tenga una Graphical User Interface (GUI).

Previamente comenzamos evaluando los diferentes lenguajes de programación y frameworks que vamos a utilizar. Elegimos el lenguaje C ya que es muy flexible, nos resulta bastante familiar y en principio, nos otorgará una disminución del tiempo de ejecución. A continuación , valoramos que framework deberiamos usar para realizar la intefaz gráfica y nos decantamos por QT por su comodidad a la hora de realizar el diseño, porque es desarrollada como un software libre y de codigo abierto y funciona en las principales plataformas. Seguidamente se trabaja de forma conjunta en la programación de nuestra herramienta software, haciendo pruebas en Julia y comparando que los resultados sean los correctos. Y una vez que hemos conseguido un programa estable y funcional, comenzamos con la programación de la interfaz gráfica.
Finalmente, se reflexiona sobre los datos obtenidos y se obtienen las conclusiones correspondientes ha dicho trabajo.

1.3. Estructura del documento

La memoria está dividida en los siguientes capítulos:

- El Capítulo Capítulo 1 contiene una introducción del proyecto, donde se hace una breve exposición del tema que vamos a tratar. Se explica la idea principal y se marcan los objetivos con el fin de cumplirlos a lo largo de la realización de este proyecto.
- El Capítulo Capítulo 2 y el Capítulo Capítulo 3 se adentran más en los fundamentos matemáticos y en las reglas en las que se basa el proyecto.
- En el Capítulo Capítulo 4 se habla de las disintas herramientas y frameworks que se han usado para la realización del mismo. Se exponen las modificaciones que se han producido con respecto a la versión anterior [1] y las mejoras que conllevan. También habla de los problemas que han surgido a lo largo del período de desarrollo y finalmente

cuenta los distintos pasos que se han realizado para la construcción del software y la funcionalidad de la interfaz.

- El Capítulo Capítulo 5 explica los resultados obtenidos por la herramienta. Se toman dichos resultados estimados y se comparan con los reales. También se comenta la mejoría en tiempos que se produce con este nuevo software.
- En el Capítulo Capítulo 6 se exponen las conclusiones obtenidas después de desarrollar nuestra herramienta, comparándolas con los objetivos y requisitos iniciales del proyecto.

Capítulo 2

Fundamentos matemáticos

Una buena estrategia para caracterizar sistemas con MCUs, comienza conociendo profundamente las propiedades de su sistema opuesto: Un sistema ideal donde solo ocurren SBUs.

Supongamos que estamos comprobando una SRAM con N bits de dirección y con un ancho de datos de N_W bits.

El experimento consiste en escribir un patrón conocido en la SRAM, irradiarlo, y finalmente leer su contenido desde la dirección más baja (0) a la más alta ($L_N = 2^N - 1$). Se asume que las direcciones donde ocurren bitflips, produciendo N_E direcciones afectadas, se almacenan en un archivo de logs. Se deben aceptar dos hechos: en primer lugar, solo pueden ocurrir SBUs. En otras palabras, las partículas inducen a un único bitflip o a ninguno. En segundo lugar, las direcciones con bitflips son distribuídas entre 0 y L_N de manera aleatoria y son equiprobables. Considerando sólo las direcciones y no su contenido interno, se obtiene un conjunto de N_E elementos, que no están repetidos y están distribuídos entre 0 y L_N.

2.1. Estadísticas para la resta positiva, XOR, etc.

Contamos con un conjunto A que contiene todos los números naturales entre 0 y L_N. Por lo tanto, contiene $L_N + 1$ elementos. A continuación, en este conjunto se define una operación binaria $d : A \times A \to A$ que cuenta con las siguientes propiedades:

1. Simetría: $d(a, b) = d(b, a)$
2. $d(a, a)$ no está definido para ningún $a \in A$

Los ejemplos son la resta positiva (P.S.), $d(a, b) = max(a, b) - min(a, b)$, la operación XOR, $d(a, b) = a \oplus b$, etc... Ambos con la prohibición de combinar dos elementos idénticos. Ahora, contamos con un conjunto nuevo llamado DA, asociado con A, tal que:

$$DA = d(a_i, a_j), \forall a_i, a_j, \backslash a_i < a_j, a_{i,j} \in A \tag{2.1}$$

La tabla 2.1 muestra el conjunto DA para $N = 3$ ($L_N = 2^3 - 1 = 7$) de la resta positiva.

	0	1	2	3	4	5	6	7
0	...	1	2	3	4	5	6	7
1	...	...	1	2	3	4	5	6
2	...	...	...	1	2	3	4	5
3	...	...	...	...	1	2	3	4
4	...	...	...	...	...	1	2	3
5	...	...	...	...	...	...	1	2
6	...	...	...	...	...	...	...	1
7	...	...	...	...	...	...	...	...

Tabla 2.1: Matriz DA para N=3 y resta positiva

A través de este ejemplo podemos observar algunas características interesantes:

- El número 0 nunca aparece entre los 8 valores posibles debido a la prohibición de restar dos valores iguales.

- Este conjunto tiene 28 elementos y 7 posibles valores. Obviamente, varios valores están repetidos pero el número de veces que aparecen no es el mismo.

En efecto, se pueden deducir dos factores fundamentales del principio de la inducción matemática: Primero, si A contiene M elementos, $([0, 1, ..., M-1])$, entonces contamos con N_{DA} elementos en DA:

$$N_{DA} = \binom{M}{2} = \frac{1}{2} \cdot M \cdot (M-1) \tag{2.2}$$

Cada elemento se crea combinando dos elementos diferentes de A, sin repetición e independiente del orden, un conocido problema en combinatoria. Tenemos entonces $M = 2^N = L_N + 1$.

Segundo, el número de veces que aparece en DA un elemento $k \in [1, 2, ..., M-1]$ es:

$$N_k = M - k \tag{2.3}$$

El enfoque clásico de la probabilidad establece que la probabilidad de ocurrencia de un evento es el número de casos favorables dividido entre el número total de casos posibles. Por lo tanto, la probabilidad de obtener $d(a_1, a_2) = k$, $0 < k < L_N$ habiendo elegido dos elementos diferentes a_1 y a_2 de A, es:

$$P_{DA}(k) = N_k / N_{DA} \tag{2.4}$$

Para cualquier operación este resultado es válido, y en el caso de la resta positiva la ecuación sería la siguiente:

$$P_{DA,PS}(k) = \frac{N_k}{N_{DA}} = \frac{2 \cdot (L_N + 1 - k)}{L_N \cdot (L_N + 1)} \tag{2.5}$$

En caso de usar la operación XOR, se puede demostrar que $P_{DA,XOR}(k) = L_N^{-1}$ *sí y solo sí* M *es un número natural potencia de 2.* De la ecuación 2.5 se puede demostrar que el

valor medio, $\overline{k}$, y la desviación típica de DA, σ, con resta positiva es $\overline{k}_{PS} = \frac{1}{3}(L_N + 1)$ y $\sigma_{PS} \approx \frac{1}{\sqrt{2}} \cdot \overline{k}$.

En cambio, si se usa XOR, los valores de los parámetros serían $\overline{k}_{XOR} = \frac{1}{2}(L_N + 1)$ y $\sigma_{XOR} \approx \frac{1}{\sqrt{3}} \cdot \overline{k}$.

Se pueden definir otras operaciones pero por lo general son expresiones difíciles con las que trabajar. Si la operación es la operación lógica AND:

$$P_{DA,AND}(k) = [L_N \cdot (L_N + 1)]^1 \cdot (3^Z - 1) \tag{2.6}$$

Siendo Z el número de ceros que tiene k en su forma binaria. Otras operaciones lógicas producen expresiones similares basadas en el número de unos o ceros.

2.2. El conjunto DA y el experimento de irradiación

Como se dijo anteriormente, obtener N_E direcciones con SBUs es formalmente equivalente a elegir de forma aleatoria N_E del rango $[0, 1..., L_N]$. De este subconjunto $V \subset A$ con N_E elementos, se genera un nuevo conjunto DV de la misma manera que se generó DA: combinando cada elemento de V con elementos mayores que él y después aplicando la operación binaria. El nuevo conjunto tendrá N_{DV} elementos:

$$N_{DV} = 0{,}5 \cdot N_E \cdot (N_E - 1) \tag{2.7}$$

La probabilidad de ocurrencias tiene como principio la ecuación 2.4, así como las versiones particulares de cada operación específica.

2.3. Parámetros estadísticos observables en DV

De acuerdo con el enfoque clásico, la probabilidad de extraer de forma aleatoria N_{DV} elementos de un conjunto A y obtener m veces un elemento k es:

$$P(k, m, N_{DV}) = \binom{N_{DV}}{m} \cdot p_k^m \cdot (1 - p_k)^{N_{DV} - m} \tag{2.8}$$

Siendo p_k la probabilidad de obtener k en un único intento.
Claramente, en el caso estudiado, $p_k \equiv P_{DA}(k)$, el número de elementos esperado para aparecer m veces es la suma de todas las probabilidades obtenidas de manera individual.

$$N_R(m, N_{DV}) = \sum_{k \in A} P(k, m, N_{DV}) = \binom{N_{DV}}{m} \cdot \sum_{k \in A} p_k^m \cdot (1 - p_k)^{N_{DV} - m} \tag{2.9}$$

Respecto a P.S., si reemplazamos en la ecuación 2.5 p_k por $P_{DA,PS}(k)$ es posible calcular $N_{R,PS}$. Si $N_{DV} << L_N$, es probable obtener una expresión simple y bastante exacta para N_R:

$$N_R(m, N_{DV}) \approx \frac{2^m}{m+1} \cdot \binom{N_{DV}}{m} \cdot L_N^{1-m} \cdot (1 + \frac{3 \cdot m - 2 \cdot N_{DV}}{L_N + 1}) \cdot (1 + \frac{2(N_{DV} - m)}{(L_N - 2) \cdot (m + 2)}) \tag{2.10}$$

Merece la pena comparar esta expresión con la derivada de la operación XOR, la cual es exacta y mucho más simple:

$$N_{R,XOR}(m, N_{DV}) = \binom{N_{DV}}{m} \cdot \frac{(L_N - 1)^{N_{DV} - m}}{L_N^{N_{DV} - 1}} \tag{2.11}$$

2.4. Validación del modelo ONLY-SBUs

2.4.1. Tests Monte Carlo

El estudio Monte Carlo[1] se realizó para validar las ideas desarrolladas en el Capítulo 2. Fueron seleccionadas 100 direcciones de un grupo de 2^{21} valores para generar DVs de 4950 elementos usando $XOR(XORDV)$ y la resta positiva $(PSDV)$. Se evaluaron el valor medio $(\overline{x})$ y la desviación típica (σ). Este proceso se repitió 1000 veces para obtener el valor medio de cada parámetro. Los resultados se muestran en la tabla 2.2, la cual confirma la concordancia entre la teoría y las simulaciones.

	XORDV		RPDV	
	Meas.	Theor.	Meas.	Theor.
	0.4998...	0.5	0.3332...	0.3333...
	0.2887...	0.2887...	0.2349...	0.2357...
Elements appearing...				
Once	4938.24	4938.33	4934.49	4934.39
Twice	5.878	5.827	7.739	7.760
Three times	0.001	0.005	0.01	0.009

Tabla 2.2: Comparación entre las simulaciones Monte Carlo y las predicciones teóricas

Otra propiedad interesante deducida de la ecuación 2.5, está relacionada con el número de unos mostrados en los elementos de DV cuando se expresan de forma binaria (llamados $t$raza).

Si se usa la resta positiva es posible determinar esto, el número esperado de elementos en el conjunto DV conteniendo m unos en su forma binaria es:

$$N_1(m, N, N_{DV}) \approx \frac{N_{DV}}{2^{N-1}} \cdot \binom{N-1}{m} \tag{2.12}$$

En el caso de usar XOR, el resultado equivalente es:

[1]Los cálculos reflejados en este documento se realizaron en el lenguaje Julia

$$N_{1,XOR}(m, N, N_{DV}) \approx \frac{N_{DV}}{2^{N-1}} \cdot \binom{N}{m} \tag{2.13}$$

Y, siguiendo un procedimiento equivalente, es posible deducir que en el caso de usar AND, el número de elementos con m unos en el conjunto DV derivado es:

$$N_{1,AND}(m, N, N_{DV}) \approx \frac{N_{DV}}{4^{N}} \cdot \binom{N}{m} \cdot 3^{m} \tag{2.14}$$

En las simulaciones Monte Carlo estos conjuntos DVs se construyeron 1000 veces, extrayendo 100 direcciones del conjunto $0, 1, ..., 2^{21} - 1$.

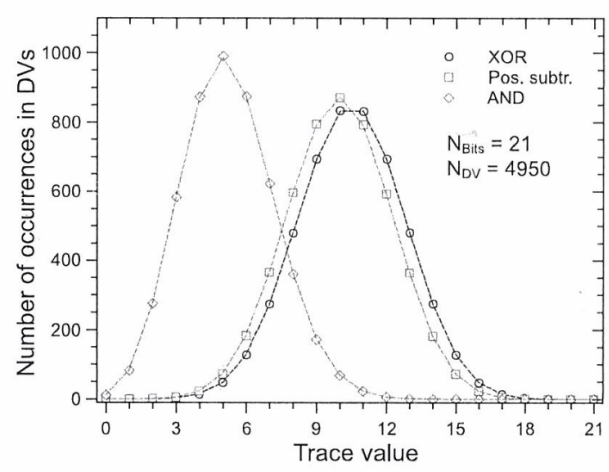

Figura 2.1: Número promedio de ocurrencias de elementos con k unos en formato binario para diferentes conjuntos DV después de 1000 simulaciones.

La figura 2.1 compara las predicciones teóricas de las ecuaciones 2.12-2.14 con los resultados obtenidos de las simulaciones. La concordancia es casi perfecta.

2.4.2. Resultados de los experimentos actuales

En la subsección anterior, las predicciones teóricas para los sistemas SEU-only han sido respaldadas por simulaciones, es obligatorio comprobar las predicciones del modelo con datos experimentales que provengan de experimentos actuales.

				MCU size				
Test	N_E	N_{DV}	SBU	2b	3b	4b	5b	6b
090nm/0x00	131	8515	94	11	1	3	0	0
090nm/0x55	120	7140	87	13	1	1	0	0
090nm/0xFF	108	5778	81	9	3	0	0	0
130nm/0x00	115	6670	62	10	5	2	2	0
130nm/0x55	146	10585	100	13	2	2	0	1
130nm/0xFF	129	8385	81	13	2	4	0	0

Tabla 2.3: Eventos observados en experimentos previos

Desafortunadamente, los datos emitidos en los experimentos realizados con las SRAMs modernas tienen una gran fracción de bitflips no relacionados con SBUs. En un trabajo anterior[3], dos SRAMs fabricadas por Cypress Semiconductor, con tecnologías CMOS de 90 & 130 nm (CY62167EV y CY62167DV, ambas con $N = 21$, $N_W = 8$), fueron irradiadas con neutrones 14-MeV en las instalaciones GENEPI2, localizadas en el *Laboratorio de Física Subatómica y Cosmología*, en Grenoble (Francia). Estas memorias se probaron con patrones diferentes (0x00, 0x55, 0xFF) para obtener más de 100 bitflips en cada ronda de lectura. Después, mediante el uso de la informacion privada del fabricante, del volumen de errores se extrajeron los MCUs, y las direcciones restantes constituían SBUs, las cuales se usaron para construir los dos conjuntos DV: $P.S.$ y XOR. La Tabla 2.4 compara de los dos conjuntos DV los valores actuales de $\overline{x}$ y σ, con su valor en la predicción teórica, ambos con alta concordancia entre ellos.

También es interesante investigar en los conjuntos DV la abundancia relativa de los valores de traza. Esto se muestra en la figura 2.2, la cual analiza los datos obtenidos de la memoria 130-nm con el patrón 0x55, el cual es el conjunto más grande según la Tabla 2.4. Las Ecuaciones 2.12 y 2.13 coinciden perfectamente con los experimentos. Merece la pena señalar el hecho de que la ecuación 2.13 ha sido utilizada por muchos autores como una aproximación a la resta positiva.

			R.P.		XOR	
Longitud	Patrón	SBUs	$\bar{x}$	σ	$\bar{x}$	σ
90 nm	0x00	94	0.345	0.242	0.503	0.288
	0x55	87	0.330	0.235	0.499	0.285
	0xFF	81	0.342	0.240	0.502	0.286
130 nm	0x00	62	0.368	0.259	0.506	0.290
	0x55	100	0.337	0.238	0.500	0.287
	0xFF	81	0.365	0.259	0.505	0.293
Teórico			0.333	0.236	0.500	0.289

Tabla 2.4: Datos estadísticos de los conjuntos DV estudiados en unidades L_N

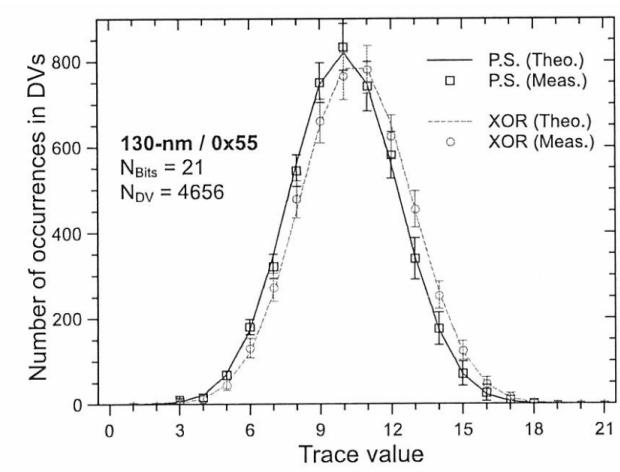

Figura 2.2: Porcentaje de elementos con valores de traza en el conjunto DV obtenidos de la resta positiva.

La figura 2.2 demuestra que esta aproximación es cercana a los resultados experimentales, pero no es tan precisa como la ecuación 2.12.

El último parámetro estadístico, pero el más importante para las aplicaciones prácticas, es el número de elementos esperados repetidos m veces en los DVs para escenarios only-SBUs, N_R. La Tabla 2.5 muestra los resultados del estudio de tres experimentos de la Tabla 2.4, que emitían el mayor número de SBUs. Conforme al estudio, la mayoría de los valores de los DVs aparecían solo una vez y algunos de ellos dos veces. Ningún valor aparecía tres o más veces.

A partir del número de ocurrencias es posible determinar el rango donde el número de valores esperados tiene un 95 % de seguridad (explicado en la ecuación 2.2) y se observa que los valores están en concordancia con las predicciones teóricas expedidas en la ecuación 2.10. Conclusiones similares se pueden deducir de la Tabla 2.6, donde para generar el conjunto DV se utilizó XOR.

	N_{DV}	Rep.	Ocur.	95 %-Conf	Theo.
130 nm	4656	1	4632	4496-4768	4642.2
		2	12	6.2-20.9	6.87
0x55		3	0	0-3.7	0.008
90 nm	4186	1	4178	4049-4307	4174.8
		2	4	1.1-10.2	5.55
0x00		3	0	0-3.7	0.005
90 nm	3655	1	3651	3530-3772	3646.5
		2	2	0.2-7.2	4.23
0x55		3	0	0-3.7	0.004

Tabla 2.5: Repetición de elementos en los conjuntos DV para sistemas ONLY-SBUs con la Resta Positiva

	N_{DV}	Rep.	Ocur.	95 %-Conf	Theo.
130 nm	4656	1	4644	4507-4780	4645.7
		2	6	2.2-13.1	5.16
0x55		3	0	0-3.7	0.004
90 nm	4186	1	4174	4045-4303	4177.6
		2	6	2.2-13.1	4.17
0x00		3	0	0-3.7	0.003
90 nm	3655	1	3655	3534-3776	3648.6
		2	0	0-3.7	3.18
0x55		3	0	0-3.7	0.002

Tabla 2.6: Repetición de elementos en los conjuntos DV para sistemas ONLY-SBUs con la operación XOR

Capítulo 3

Reglas propuestas para descubrir valores DV anómalos e identificar MCUs

Dado que las ecuaciones detalladas en el Capítulo 2 son apropiadas para describir un escenario sólo con SBUs, las desviaciones observadas en los experimentos respecto a sus previsiones son indicios de la presencia de MCUs.
Es necesario definir un criterio para diferenciar causalidad de aleatoriedad. En este documento, en el Capítulo 2, se postula que un fenómeno observado no puede ser atribuído a la aletoriedad si el número de ocurrencias pronosticado es menor que $\epsilon_R = 0{,}05$. Este umbral se basa en el éstandar de 95 % de seguridad usado en muchos campos de la física, y significa que el fenómeno ocurre en 1 experimento de cada 20. El valor de ϵ_R es una decisión personal de los autores, por lo que otros investigadores que sigan este método pueden ajustar el valor disminuyéndolo si buscan cálculos más estrictos.

3.1. Repeticiones excesivas y autoconsistencia

Supongamos que hemos irradiado una memoria y en N_A direcciones diferentes reportamos bitflips. Utilizamos este conjunto de direcciones para crear conjuntos DV con XOR y $P.S.$, y evaluamos el número de veces que cada valor entre 0 y L_N se repite en cada conjunto DV. Ya que L_N y N_{DV} son conocidos, los cálculos númericos de las ecuaciones 2.10 y

2.11 permiten determinar el valor umbral de m (m_0) desde el cual, el número esperado de elementos repetidos es menor que ϵ_R. En consecuencia, los elementos que aparecen en los conjuntos DV más de m_0 veces no son compatibles con los sistemas only-SBU y se deben atribuir a la ocurrencia de eventos múltiples.

Aunque los valores anómalos de DV solo vinculan dos direcciones, se pueden recrear MCUs de un tamaño mayor. Así, si dos direcciones A_1 & A_2 están enlazadas por un valor anómalo DV_0, pero A_1 también está enlazada con otra dirección A_3 con otro valor DV_1, es evidente que ambos pares deben ser fusionados formando un MCU que involucra las tres direcciones: $\{A_1, A_2\} \cup \{A_1, A_3\} = \{A_1, A_2, A_3\}$

Desafortunadamente, no todos los valores son apropiados para detectar eventos múltiples debido a un interesante fenómeno: la interacción entre eventos múltiples. Este fenómeno se explica en la Figura 3.1: cuando las direcciones de las celdas de dos MCUs se combinan para crear el conjunto DV, la ocurrencia de los vectores de desplazamiento respecto a las celdas se ve incrementada de forma anómala.

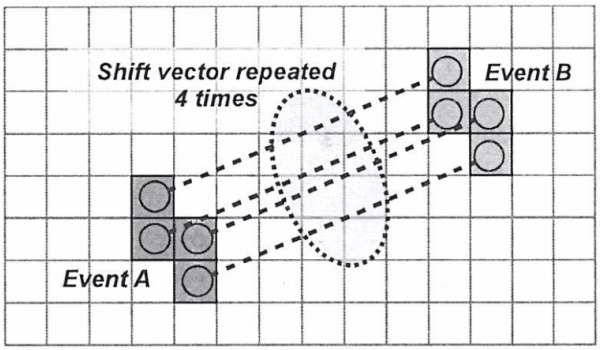

Figura 3.1: Interacción entre MCUs. Cada cuadrado simboliza una celda de memoria. La interacción entre MCUs distantes y con la misma forma puede interpretarse como valores repetidos de manera anómala en el conjunto DV.

Para evitar este problema, en trabajos anteriores[3] se propuso el no tomar nunca más de 15 valores. Esta figura se obtuvo después de realizar algunos tests prueba-error pero depende en gran medida de la voluntad del investigador. En el documento actual, se propone un nuevo camino para evitar la selección de falsos indicadores de MCUs sin la intervención de investigadores. Se ha llamado a esta idea *"self-consistency"*. En la Figura 3.1, se puede observar que los vectores de desplazamiento aparecen 4 veces, exactamente el tamaño de los MCUs. Esto proporciona una pista para desestimar falsos positivos: los valores repetidos de forma anómala son útiles, sí y sólo si, están repetidos más veces que el tamaño del MCU propuesto más grande.

Esta regla es válida tanto para XOR como para R.P.. La regla es iterativa. En primer lugar, se selecciona el elemento más repetido en el conjunto DV y se identifican las direcciones involucradas. Después, se selecciona el segundo elemento más repetido, se añadide al conjunto preliminar de valores DV anómalos, y se reclasifican las direcciones. Este proceso continúa hasta que se viola el principio "self-consistency".

Esta regla es democrática. Esto significa que si varios elementos DV que aparecen el mismo número de veces, todos ellos deberían ser añadidos como un todo al conjunto preliminar de valores DV anómalos o no añadir ninguno. También, si el principio "self-consistency" se viola después de agregar los elementos, todos ellos deberán ser eliminados. Además, esta regla puede ser aplicada independientemente de que sean conjuntos DV XOR o $P.S.$ Una vez que sean procesados ambos conjuntos, es obligatorio combinar los resultados para obtener el conjunto más grande posible de pares de direcciones.

Vamos a ejemplificar el procedimiento con los resultados del test 130nm/0x00 del Cuadro 2.3, donde 115 direcciones se han visto afectadas por errores. En este caso, $N_{DV} = 6670$. Por lo tanto, la ecuación 2.11 predice que en el conjunto $XORDV$ aparecen 2 veces ~ 10.57 elementos, y aparecen 3 veces 0.011 elementos. Para la resta positiva, la ecuación 2.10, predice 14.07 y 0.022 elementos. Usando el umbral σ_R no debería haber eventos repetidos 3

o más veces en ambos conjuntos de DV. De todas formas, las predicciones no concuerdan con los datos experimentales, como muestra la Tabla 3.1. Por lo tanto, tiene que haber algo además de los SUBs: la existencia de eventos múltiples.

XOR		Resta Positiva	
Valor	Ocurrencias	Valor	Ocurrencias
0x010001	22	0x0FFFF	14
0x010101	14	0x000100	13
0x000100	13	0x010001	8
25 elementos	4	0x0FEFF	5
		0x25B7E 0x09C382	4
90 elementos	3		
		13 elementos	3

Tabla 3.1: Elementos en exceso en el conjunto DV para 130-NM/0x00

El concepto "self-consistency" será aclarado paso a paso en la Tabla 3.2, para los resultados que implican a la resta positiva en el Tabla 3.1. Inicialmente, el conjunto de valores anómalos usados para la clasificación está vacío y los elementos más repetidos son añadidos tras sucesivos pasos. Después de cada paso, se organizan los bitflips (Columnas $SEUs$ *(tamaños de eventos vs. apariciones)*) y se verifica la "self-consistency". Sólo en la quinta etapa algo va mal: Se propone un MCU de 8-bits, pero los valores anómalos que fueron añadidos solo aparecen cuatro veces. Por consiguiente, se viola el principio "self-consistency". Como consecuencia, estos elementos se deben eliminar del conjunto de valores anómalos, se debe recalcular conjunto de MCUs y el procedimiento debe detenerse. La regla de "self-consistency" resultó ser más preciso que la primera propuesta de no coger más de 15 valores [1]. Por ejemplo, en el caso de 130 nm/0x00, $0x25B7E$ y $0x9C382$ habrían sido aceptadas a pesar del hecho de que no se corresponden con direcciones cercanas entre ellas y por tanto no pueden corresponder a un mismo MCU.

			Tamaños de los eventos vs. ocurrencias						
Step	Valor	Rep.	1	2	3	4	5-7	8	S.C.?
1	0x0FFFF	14	87	14	0	0	0	0	Sí
2	0x00100	13	73	11	4	2	0	0	Sí
3	0x10001	8	66	9	5	4	0	0	Sí
4	0x0FEFF	5	66	9	5	4	0	0	Sí
5	0x25B7E 0x9C382	4	66	9	5	0	0	1	NO

Tabla 3.2: Aplicación de la regla self-consistency para 130-NM/0x00

La técnica de clasificación de SBU/MCU propuesta en este documento combina las clasificaciones de rendimiento tanto de XOR como de la resta positiva. La clasificación combinada es significativamente más precisa que las clasificaciones de forma separada. Así, por ejemplo, en el caso del test de 90 nm/0x55 los valores DV anómalos de XOR permitieron clasificar 120 direcciones en 96 SBUs y 12 MCUs de 2-bits. La resta positiva produjo 99 SBUs, 7 MCUs de 2-bits, 1 MCU de 3-bits, y 1 MCU de 4-bits. Después de la combinación, la propuesta fue mucho más precisa: 89 SBUs, 12 MCUs de 2-bits, 1 MCU de 3-bits, y 1 MCU de 4-bits si comparamos con los valores reales mostrados en la Tabla 2.3.

3.2. Combinación de datos de diferentes experimentos

A veces, los experimentos se repiten en condiciones idénticas o similares debido a diferentes razones: la verificación de la repetición del test, cambios en el tipo de partícula o su energía, etc. En esta situación, los datos que provienen de distintos resultados pueden ser combinados para mejorar la clasificación de los eventos.

Supongamos que vamos a realizar varios experimentos sobre una memoria obteniendo diferentes conjuntos de direcciones con bitflips. Les llamaremos A, B, C, etc. Es inmediato que los valores DV críticos, tanto para XOR como para la resta positiva, sean válidos para otros experimentos desde que estos valores son solo relacionados con la estructura interna

de la SRAM y no con el tipo de radiación. La unión de los diferentes conjuntos de valores anómalos puede usarse para reclasificar los bitflips de cada uno de los conjuntos de direcciones aislados. Por ejemplo, en el caso de la memoria de 130-nm, los siguientes valores anómalos se observaron en la resta positiva: {0x256, 0x65279, 0x65535, 0x65537} del patrón {0x00, 0x65535, 0x524032, 0x589567} del 0x55, y {0x65535, 0x65537, 0x262400, 0x524032} de 0xFF. Claramente, la unión de estos conjuntos, {0, 256, 0x65279, 0x65535, 0x65537, 0x262400, 0x524032, 0x589567} es mejor que los conjuntos separados parcialmente.

Todavía hay otra manera de mejorar los resultados combinando resultados de distintos experimentos si estos se realizan en condiciones idénticas. Debemos tener en mente que el tratamiento de datos se basa en la suposición de que datos aleatorios son recogidos de un conjunto DA enorme. No importa como se seleccionan los datos.

De hecho, en el Capítulo 2 no está postulada la manera de seleccionar los datos. Por lo tanto, si seleccionamos N_{DV1} elementos en una primera ronda y N_{DV2} elementos en la siguiente, la unión de los dos conjuntos es un nuevo conjunto de $N_{DV1} + N_{DV2}$ elementos elegidos de manera aleatoria que cumplen las condiciones del Capítulo 2. El histograma con el número de ocurrencias relacionado con este nuevo conjunto es la adición de los histogramas parciales.

La principal ventaja de combinar datos de diferentes experimentos es que se mitiga la posibilidad de confundir las fluctuaciones aleatorias, con la presencia de valores DV anómalos. La teoría estadística muestra que si el número de repeticiones en un único experimento se distribuye entre $m \cdot (1 \pm \Delta\alpha)$, la suma de n experimentos similares nos dirigirá a $\sim n \cdot m \cdot (1 \pm \Delta\alpha/\sqrt{n})$. Mientras tanto, el número de ocurrencias DV anómalas es proporcional a n por lo que son más fáciles de localizar.

También, la abundancia de valores DV anómalos causada por la interacción entre MCUs se mitiga si se combinan datos de diferentes experimentos.

3.3. La regla del patrón

La ocurrencia de valores DV anómalos relacionados con MCUs tienen una fuerte dependencia con el patrón de escritura. Los elementos DV difíciles de reconocer en un experimento pueden ocurrir varias veces en otras rondas después de cambiar el patrón. Por lo tanto, se aconseja el uso de varios patrones en experimentos diferentes para sacar a la luz estos valores. Después, los conjuntos diferentes descubiertos de valores DV anómalos se unen para clasificar las direcciones en pares.
Sin embargo, en este caso, donde ambas operaciones (XOR y R.P.) están involucradas, se ha descubierto que es contraproducente unir conjuntos DV con patrones diferentes como se explica en el Capítulo 4.2. La razón es que algunos elementos con valores anómalos aparecen muchas veces con un patrón pero pocas con los demás patrones, de tal manera que así la suma de ocurrencias se compensa, haciendo imposible discernir el exceso de ocurrencias con fluctuaciones aleatorias.

Por lo tanto, el conjunto de datos experimental se debe dividir en subconjuntos más pequeños según el patrón y estos se deben estudiar independientemente.

3.4. La regla de la traza

Las memorias típicas se diseñan modularmente usando bloques de unidades (unit blocks) que se multiplexan usando los bits de dirección. Esto significa que las celdas adyacentes normalmente comparten una gran parte de los bits de dirección. Cuando a estas direcciones se les aplica la operación XOR, el elemento DV resultante tendrá muchos ceros y muy pocos unos en su forma binaria. Por ejemplo, en la memoria 90-nm, la regla de auto-consistencia emitía 0x00002, 0x00006, 0x0C000, 0x0E000 con no más de 3 unos en 21 bits. Por lo tanto, la abundancia anómala de elementos con un bajo valor de traza en el conjunto $XORDV$

señala la ocurrencia de MCUs.

La idea de esta regla es simple: es necesario buscar los elementos con un bajo valor de traza en el conjunto $XORDV$ y verificar si estos elementos aparecen más de lo esperado. En nuestros experimentos decidimos buscar elementos con una traza menor o igual que 3 y que aparecieran $m0$ o más veces en el conjunto $XORDV$. ($m0$ fue definido en el Capítulo 4.1.)

3.5. La regla de XOR

Otra consecuencia interesante de la organización modular de muchas SRAMs es que se pueden extraer nuevos valores $XORDV$ anómalos realizando la operación XOR sobre otros valores $XORDV$ confirmados. En Capítulo 3 de este documento, el siguiente escenario fue despreciado: Después de un experimento, fue construído un conjunto DV con un conjunto de valores $XORDV$ anómalos, $AXORDV$, después de aplicar unas reglas previas. Entonces, si un elemento $K \in XORDV$, $K \notin AXORDV$, y ocurre una de las siguientes tres condiciones:

- Puede expresarse como $K = C_1 \oplus C_2$ con $C_1, C_2 \in AXORDV$
- $K \oplus C_1 = C_2$ con $C_1, C_2 \in AXORDV$
- $K \oplus K_2 = C_1$ con $C_1 \in AXORDV$, y $K_2 \in XORDV$

Entonces, el elemento K es candidato de ser otro elemento a añadir al conjunto $AXORDV$. El problema en el supuesto inicial de la regla era que no hay una manera de determinar si la ocurrencia se acababa de emitir por aletoriedad, o por el contrario, era una secuela de la estructura interna. Por lo tanto se propone sólo aceptar aquellos elementos descubiertos con estas operaciones pero que también aparecen m_0 veces o más. Además, la aplicación de la regla en varios conjuntos de experimentos muestra que la tercera regla generalmente es improductiva y requiere mucho tiempo de cómputo.

3.6. La regla de los MCUs preliminares

En subsecciones previas, se han propuesto varias reglas para identificar pares de direcciones relacionadas por valores DV anómalos. Esto permite agrupar las direcciones en SBUs o MCUs de multiplicidades diferentes.

En un MCU p-bit, es posible determinar $0{,}5 \cdot p \cdot (p-1)$ valores DV que relacionan celdas cercanas para cada operación. Así, por ejemplo, en un MCU 4-bit es posible calcular 6 valores $XORDV$ y los otros 6 valores $PSDV$ que provienen de las direcciones involucradas. Es posible que después de aplicar reglas previas, algunos de estos valores $XORDV$ o $PSDV$ no se hayan descubierto (por ejemplo, como consecuencia de que previamente haya sido violada la regla de auto-consistencia). Por lo que esta regla propone incluirlos en el conjunto de valores $XORDV$ o $PSDV$ anómalos, si estos aparecen más de m_0 veces.

En consecuencia, después de una identificación preliminar de los MCUs, estos deben analizarse a fondo para acumular más valores DV críticos. Desafortunadamente, la aplicación de esta regla en los conjuntos irradiados del conjunto $PSDV$ nos conduce a falsos positivos que relacionan direcciones que no son cercanas físicamente. Esta regla solo funciona correctamente en la detección de elementos nuevos en $XORDV$.

Esta regla es iterativa. Después de cada búsqueda de nuevos valores $XORDV$, la organización de los eventos múltiples puede cambiar: aparición de nuevos MCUs 2-bit, crecimiento de algunos MCUs, unión de dos MCUs pequeños para producir uno más grande... Después de que se identifica un nuevo MCU, los nuevos valores $XORDV$ potenciales se deben analizar hasta que no se descubran nuevos elementos. Una vez sucede esto, termina la ejecución de esta regla.

A continuación, se muestra el algoritmo desarrollado para la extracción de MCUs.

Algorithm 1 Proposed methodology to extract MCUs

Input: of affected addresses: $Addr_1, Addr_2, ..., Addr_n$
Output: Set of MCUs: MCUs
Iterate on the input sets of addresses $Addr_i$ obtained in several rounds of reading, by checking different patterns (Pattern Rule)...
for each Addr **do**
  #Step 1: Create DV_s and initialize MCUs set.
  XORDV = CreateXORDV($Addr$);
  PSDV = CreatePSDV($Addr$);
  MCUs = θ;
  #Step 2: Study od DVs by applying Equation (10)
  $XCandidates = InExcess(XORDV, 0{,}05)$
  $Candidates = InExcess(PSDV, 0{,}05)$
  #Step 3: Apply Self-consistency in XORDV
  AXORDV = θ
  $violatedSelfConsistency = FALSE$;
  while $violatedSelfConsistency == FALSE$ **do**
    $newCandidates = getMostRepeated(XCandidates)$;
    updateSet(&XORDV, newCandidates);
    $newCandidates = getMostRepeated(XCandidates)$;
    $updateMCUs = updateMCUs(MCUSs, AXORDV)$;
    if ($numberOfOcurrences(NewCandidates, XORDV)>$
      sizeLargestMCU($UpdatedMCUs$) **then**
      $MCUs = updatedMCUs$;
    else
      violatedSelfConsistency = TRUE;
    end if
  end while
  #Step 4: Self-consistency in PSDV: Similar to Step 3
  #will create the set APSDV and update MCUs again
  #Step 5: Trace rule in XORDV
  $newCandidates$ = extractLowTrace(XCandidates);
  updateSet ($\&AXORDV, newCandidates$);
end for
#Step 6 : Xoring rule
newCandidates = XORRule(AXORDV, XCandidates);
updateSet(&AXORDV, newCandidates);
#Step 7 : Update MCUs with AXORDV and PSDV sets
$MCUs = updateMCUs(MCUs, AXORDV)$;
$MCUs = updateMCUs(MCUs, APSDV)$;
`Step 8 : Preliminary MCU Rule`
for each MCU_i in MCUs **do**
  allXValues = extractXValues(MCUs); 29
  updateSet(&AXORDV, allXValues, XCandidates);
end for
MCUs = updateMCUs(MCUs, AXORDV);
return$MCUs$;

Capítulo 4

Herramienta

Nuestra herramienta consta de dos partes: un programa que aplica el algoritmo explicado anteriormente sobre unos archivos elegidos que contienen datos para obtener los SBUs y los MCUs, y una interfaz gráfica que ejecuta el programa mencionado anteriormente y que dibujando el hardware muestra los puntos exactos de la memoria donde han sucedido dichos eventos.

4.1. Entorno de desarrollo

4.1.1. Entorno software

Para la realizacón de este proyecto nos hemos apoyado fundamentalmente en dos plataformas para la implementación del programa. Eclipse, una plataforma de integracion de herramienta para el desarrollo de codigo abierto, y xCode, otro entorno de desarrollo integrado exlcusivo para macOS. Ambas plataformas soportan el lenguaje C, el cual usaremos para el programa principal.

Para la realización de la interfaz gráfica por otra parte, hemos usado en este caso un framework multiplataforma orientado a objetos denominado Qt, utilizando como lenguaje de programación C++ ya que es su lenguaje de programación nativo.

4.1.2. Eclipse

Eclipse es un entorno de desarrollo integrado *(Integrated Development Environment, IDE)* compuesto por un conjunto de herramientas de programación de código abierto multiplataforma. Consta de un workspace sobre el que trabajar y un amplio conjunto de plug-ins que nos permiten personalizar el entorno. Aunque nosotros lo hayamos utilizado para escribir un programa en C, esta plataforma ha sido tipicamente usada para desarrollar entornos de desarrollo integrados (IDE), como el IDE de Java (JDK). También, mediante la instalación de plug-ins permite desarrollar código en Ada, ABAP, C, C++, C#, COBOL, D, Fortran, Haskell, JavaScript, Julia, Perl, PHP, Python, R, Ruby... Eclipse posee una licencia denomianda Eclipse Public License, una licencia de software libre pero que es iincompatible con la Licencia pública general de GNU.

4.1.3. xCode

xCode es un entorno de desarrollo integrado para el sistema operativo macOS que contiene una serie de herramientas para desarrollo de software desarrolladas por Apple.

Xcode también cuenta con compiladores del proyecto GNU (GCC), y permite compilar código de diferentes lenguajes: C, C++, Swift, Objective-C, Objective-C++, Java... Cuenta con una herramienta que permite la creación de interfaces de usuario, Interface Builder, pero nosotros decidimos usar Qt al parecernos más intuitiva y amigable.

4.1.4. Qt

Qt es un framework multiplataforma orientado a objetos. Su utilidad principal es el desarrollo de programas que requieran de interfaz de usuario.

Es muy cómoda a la hora de diseñar interfaces gráficas porque cuenta con un modelo de programación que te permite añadir objetos o widgets de una manera tan sencilla como

simplemente seleccionar el objeto y arrastrarlo al lugar en el que lo quieres colocar. Para implementar relaciones entre objetos se crean señales que los comunican dependiendo de la acción que se realice en cada uno de ellos. Qt utiliza el lenguaje de programación C++ de forma nativa, pero también puede ser utilizado con otros lenguajes de programación haciendo referencia a otras bibliotecas. Qt es desarrollada como un software libre y de código abierto y está distribuida bajo los término de GNU LGPL

4.2. Desarrollo

Para el desarrollo del proyecto no hemos partido de cero, hemos continuado con la ultima version del proyecto presentado en [1] por nuestros directores entre otros autores, cuyo apoyo, indicaciones y dedicación, ya sea a traves del documento, el programa inicial, como de reuniones , nos ha facilitado el trabajo.

El proyecto se apoya en un programa desarrollado en lenguaje julia, el cual carece de una interfaz grafica con la que interaccionar y se ejecuta por consola, haciendo que sea menos atractiva para el usuario a la hora de trabajar, más dificil de operar con ella y hay que tener presente la funcionalidad de los comandos. Con el fin de mitigar esta carencia, hemos optado por utilizar un framework multiplataforma orientado a objetos denominado QT, para desarrollar la parte grafica, la cual facilitara la interaccion del usuario con el sistema de una forma más sencilla.

4.2.1. Julia

Julia es un lenguaje de programación homoicónico, multiplataforma y multiparadigma de tipado dinámico de alto nivel y alto desempeño para la computación genérica, técnica y científica, con una sintaxis similar a la de otros entornos de computación similares. Dispone de un compilador avanzado (JIT), mecanismos para la programación en entornos distribuídos

y de una amplia biblioteca de funciones matemáticas. Esta biblioteca está desarrollada en Julia, pero también contiene código desarrollado en C o Fortran.

4.2.2. C

C es un lenguaje de programación orientado a la implementación de Sistemas Operativos, concretamente Unix, nacido en 1972. El código producido en este lenguaje es muy eficiente, por lo que aunque el lenguaje sea “antiguo"sigue siendo muy utilizado para crear aplicaciones. Se trata de un lenguaje de tipos de datos estáticos, débilmente tipificado, de medio nivel, porque cuenta con componentes como las estructuras de alto nivel, y también permite construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen brindar extensiones al lenguaje que permiten la mexcla de código C con código ensamblador, el acceso directo a memoria o a dispositivos de entrada y/o salida.

Si los programas creados en este lenguaje siguen sus estándares, el código es totalmente portable entre plataformas y arquitecturas.

Ventajas:

- Es un lenguaje muy eficiente gracias a que tiene la posibilidad de realizar implementaciones de bajo nivel.
- Existen compiladores para prácticamente todos los sistemas, por lo que es altamente portable.
- Cómodo para la realización de programas modulares y utilizar código de bibliotecas existentes.
- Disminución del tiempo de ejecución.

Inconvenientes:

- Velocidad de desarrollo por parte del programador dada la complejidad del lenguaje.
- Reserva y liberación de memoria, ya que C no cuenta con un gestor automático que facilite este proceso.
- Mantenimiento de los programas más costoso y difícil.
- El uso de punteros y direcciones de memoria requiere tiempo y es un proceso delicado.

4.3. Modificaciones

El primer paso a la hora de abordar este proyecto es realizar una fase de analálisis en la cual nos de familiarizarnos con las diferentes técnicas existentes para discernir los errores y definiremos los contenidos y funcionalidades.Estas tecnicas se encuentra en lenguaje Julia y debemos adaptarnos al nuevo lenguaje y aprender a usar la plataforma propia de Julia. Consta de una consola propia desde la cual se ejecuta el programa y arroja los distintos resultados.

Una vez nos adaptamos a este lenguaje, comienza el proceso de creación de nuestro propio programa. Para ello establecemos una fase de diseño, en la que decidimos el conjunto de estructuras que representarán los datos necesarios y el detalle procedimental del programa, todo ello con el objetivo de facilitar nuestro trabajo y mejorar la ordenación de los archivos. El proyecto contiene tres archivos, divididos en un programa principal *main.c*, el cual tendra las llamadas a las principales funciones y ademas generara los distintos archivos con los resultados y conclusiones finales. También, creamos otro archivo *lib.h* el cual hace de biblioteca y contiene todas y cada unas de las funciones especificas implementadas para la obtencion de los calculos mencionados en el Capitulo 2. Y tambien, un archivo *Struct.h* el cual contiene las estructuras de matrices, vectores, etc, que iremos creando conforme avan-

zamos en el codigo.

Finalmente, comienza la fase de implementación, en la cual crearemos nuestro programa en codigo C y desarrollaremos las distintas funciones y algoritmos. Como fruto, obtendremos un programa fuente, que una vez compilado, dará lugar a nuestro ejecutable.

Por otro lado hay otra fase más de diseño, en la cual plantearemos cómo será nuestra interfaz grafica y qué se va a encontrar el usuario al ejecutarla. A la hora de diseñar una buena interfaz grafica enfocada al usuario es necesario tener claro los objetivos del hipertexto, teniendo en cuenta no sólo lo que se persigue ofreciendo información, sino las necesidades que van a tener los usuarios a la hora de consultarlo. Es por ello que hemos creado una interfaz sencilla, la cual cuenta con una pantalla principal en la qué podemos observar que archivo estamos ejecutando, la grafica correspondiente con las direcciones que contienen fallos y más opciones que detallaremos en el punto 4.6. Como hemos mencionado durante el proyecto, hemos desarrollado esta interfaz usando el framework Qt, el cual usa el ejecutable generado por nuestro programa principal para obtener los resultados y plasmarlos de una forma más agradable.

Con este tipo de modificaciones conseguimos una optimización del codigo, mejoras en el tiempo de ejecución del programa frente a Julia, también se disminuye el espacio de memoria utilizado con el uso de memoria dinámica, la cual se reserva en tiempo de ejecucuión y su tamaño varia en funcion de lo que requiera el programa.

4.3.1. Optimización

La optimización de código constituye un elemento importante en el desarrollo de nuestra herramienta, que proporciona mejoras considerables que permiten un mayor acercamiento

al éxito de esta. Nuestro objetivo con esta optimización radica en maximizar la eficiencia temporal, es decir, mejorar el tiempo de ejecución y espacial, mejorar el espacio de memoria utilizado permitiendo reorganizar todo el código de manera limpia, eficiente y correcta.

Hemos optimizado el código mejorando ciertas evaluaciones de expresiones y sentencias más específicas. También hemos eliminado código muerto que no se utilizaba lo cual hace que favorezca el tiempo de ejecución y espacio de memoria. Reducimos el número de variables innecesarias dentro de bucles y reutilizamos aquellas que son posibles. Se han dividido funciones en otras más específicas dependiendo del tipo de dato que procesan y devuelven, así como de la creación de otras más genéricas con el fin de no repetir código. Debido a que el programa genera y maneja un gran número de matrices de distintos rangos y dimensiones, también se ha generado un archivo structs.h en el que se definen distintas estructuras de diferentes tipos y datos que se hacen referencia a ella desde el programa principal. Gracias a esto conseguimos una mejor estructuración del código, además de una optimización a la hora de definir o usar nuevas variables, pudiendo hacer referencia a las ya creadas y almacenadas anteriormente.

4.3.2. Mejoras

En este proyecto contemplamos dos importantes mejoras principales con respecto al programa desarrollado en Julia. La primera mejora tiene que ver con el procesamiento de la información, mientras que Julia procesa la información con una gran biblioteca de funciones, en C debemos crear los distintos métodos necesarios para poder realizar las mismas funciones. Esto hace que nuestra herramienta sea más potente, ya que sólo comprobamos lo que necesitamos. Es por ello que hace que sea más eficiente ya que podemos usar sus características de bajo nivel para realizar implementaciones óptimas. Por consiguiente, ob-

tenemos una reducción muy considerable tanto en el tiempo de ejecución como en el uso de de la memoria.

Como podemos observar en la Tabla 4.1, al utilizar memorias de 90-NM conseguimos que el tiempo de cómputo del programa sea hasta 3 veces más rapido que el del programa inicial. También obtenemos una reducción en el uso de memoria considerable, logrando una disminución de más de la mitad en la mayoría de los experimentos. Para este tipo de experimentos, el uso de la memoria es muy importante, ya que al trabajar con un gran volumen de datos es fundamental administrar correctamente el uso de memoria, evitando problemas de falta de memoria.

	Julia		C	
	Tiempo s	RAM	Tiempo s	RAM
90-NM / NTS	3.099454	320 MB	0.895101	90.3 MB
90-NM / TS	3.094372	320 MB	0.685183	90 MB
130-NM / NTS	4.197029	480 MB	3.123077	172.2 MB
130-NM / TS	3.9238125	400 MB	1.928815	172.1 MB

Tabla 4.1: Comparación de tiempos y uso de la memoria en la ejecución de los distintos programas

Otra mejora que destaca considerablemente es la incorporación de una interfaz gráfica. Con ella se consigue una mejor visualización generalizada del programa. Mientras que en Julia se ejecutaba el programa a través de la consola del sistema confiriendo al usuario un medio muy poco amigable con el que trabajar, con Qt disponemos de una ventana sencilla, concisa, con efectos de relieve y botones para realizar distintas funciones, mucho más accesible, afable e intuitiva para cualquier persona que quiera interactuar y trabajar con la herramienta de una forma mucho más sencilla.

4.4. Problemas

Como es de esperar tambien hemos encontrado problemas a medida que desarrollábamos el proyecto. El primer problema tiene que ver fundamentalmente con la elección del lenguaje de programación. A pesar de las numerosas ventajas que tiene el lenguaje C, Julia cuenta con un gran numero de librerías y funciones matemáticas para la realización de los distintos cálculos matemáticos del algoritmo, los cuales no existen en C, lo que hace que utilicemos parte del tiempo en pensarlas y programarlas en vez de no preocuparnos por cómo se hacen, si no simplemente entender cómo usarlas con el fin de obtener el mismo resultado.

El lenguaje Julia cuenta con un tipado dinámico, es decir, qué no hace falta especificar que tipo de variable queremos crear, pudiendo tomar valores de distinto tipo en distintos momentos, lo cual nos dificultaba conocer los tipos. También hay que mencionar que las conversiones de tipos númericos tampoco se encuentran en C.

Otro problema surgió a la hora de integrar el código generado en lenguaje C del programa principal con la interfaz grafica, diseñada e implementada con lenguaje C++. La solución fue sencilla ya que generamos un ejecutable del programa y la propia interfaz se encarga de ejecutarlo.

4.5. Pasos del programa

El primer paso que realiza nuestro programa es la creación de una matriz de tiempos para mantener un seguimiento sobre el tiempo empleado en realizar las determinadas tareas que se llevan a cabo durante la ejecución de éste. Posteriormente, con los tiempos registrados, pueden conocerse las marcas de tiempo en las que se ha efectuado cada paso del programa. Primordialmente, su uso se limita a conocer el tiempo total de ejecución del programa.

El siguiente paso es obtener y leer los datos proporcionados por los investigadores sobre

las diferentes características del experimento y las direcciones de memoria obtenidas en éste. Estos datos se encuentran en archivos Julia o de extensión “.jl". Mediante una función se parsean todos los valores que contiene dicho archivo, obteniendo el número de bits de dirección, el ancho de palabra, el número de bits por bloque y el archivo “.csv"del que se adquieren las direcciones asociadas a los distintos tests y cuántos tests son. Además, del archivo anterior, también podrían obtenerse conjuntos de valores anómalos previamente declarados por el usuario del programa.

```
const NbitsAddress=21;
const NWordWidth=8;
const Nbits4blocks = 0;

const Pattern = [
1 0x00
2 0x55
3 0xFF
];

const PreviouslyKnownXOR=[
#0x000100
#0x010001
#0x010101
];

const PreviouslyKnownPOS=[
#256
#65535
];

Content=round(UInt32, readcsv("DATA/CY62167_130nm_TS.csv"))
```

Figura 4.1: Ejemplo de archivo .jl para experimento sobre 130nm.

Seguidamente, se comprueba que todos los datos recolectados anteriormente son correctos para seguir con la ejecución del programa. Se guarda el número de direcciones válidas de cada test y se calculan las variables L_N (rango total de las direcciones) y N_{DV} (número de elementos del conjunto DV para cada test).

A continuación, se obtiene el conjunto DV tanto para la resta positiva como para la operación XOR, y con este conjunto, se conoce el número de repeticiones de los elementos de cada test para la resta positiva y para la operación XOR. Este número de repeticiones se representa en un histograma para cada operación.

Conocidos los datos anteriores, se inicia la búsqueda de los valores con repeticiones anómalas, aplicando la regla "self-consistency.[en] todos los tests, y alcanzando todos los valores

anómalos. Luego, se aplica la regla de la traza, pero únicamente a las repeticiones anómalas y los valores anómalos obtenidos con la operación XOR. Posteriormente, si existieran valores declarados por el usuario, este sería el punto del programa donde se realizaría la unión de estos valores con los obtenidos en el punto anterior.

Después, sobre los conjuntos de datos se aplica por primera vez la regla de los MCUs preliminares, que permitirá agrupar los elementos en SBUs o MCUs de diferentes multiplicidades al obtener nuevos valores.

Se aplicará la regla de XOR sobre el conjunto de valores anómalos $XORDV$ obtenido en el paso anterior que dará lugar a otro conjunto más.

Se repite el empleo de la regla de los MCUs preliminares sobre el último conjunto de la resta positiva y el de la operación XOR para obtener finalmente los dos conjuntos definitivos, de los que se lograrán los MCUs que suceden en los distintos tests.

Para terminar, conociendo los MCUs, se podrá aplicar una función para conocer en qué test han sucedido, localizarlos en la memoria y conocer sus bitflips.

Finalmente, se arrojan como resultados de la ejecución del programa, dos archivos y un ejecutable. El primer archivo, llamado *affectedAdresses*, contiene los resultados de los MCUs detectados en los distintos test que se han realizado conteniendo la fecha y hora en la que ha sido generado con el fin de llevar un registro de las pruebas. El otro archivo denominado *results.txt*, contiene la misma información que el anterior, pero preparada para que desde la interfaz gráfica se lea con más facilidad.

4.6. Funcionalidades de la interfaz

Como bien hemos mencionado numerosas veces a lo largo de este documento, se ha creado una interfaz sencilla y amigable con el fin de que el usuario pueda interactuar de forma más sencilla con la herramienta, pudiendo ejecutarla sin problemas y sin mayores dificulta-

des.

Al ejecutar el programa, la interfaz gráfica contiene una ventana principal sencilla en la que encontramos un botón para poder elegir el archivo de datos que se quiere procesar además del correspondiente menú superior con botones de cerrar, maximizar y minimizar la ventana.

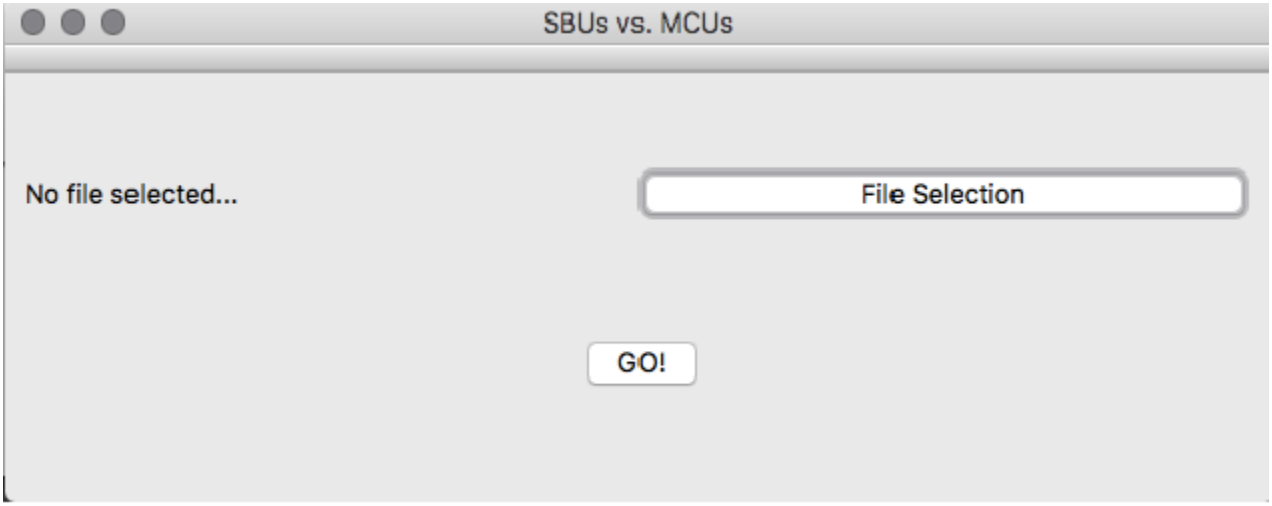

Figura 4.2: La ventana contiene un botón y un label qen el que aparecerá el archivo elegido.

Al pulsar sobre el botón de abrir archivo, aparecerá una ventana nueva que descubre el explorador de archivos por donde movernos a través de todo el directorio de nuestro ordenador y seleccionar el archivo deseado para ejecutar nuestro experimento.

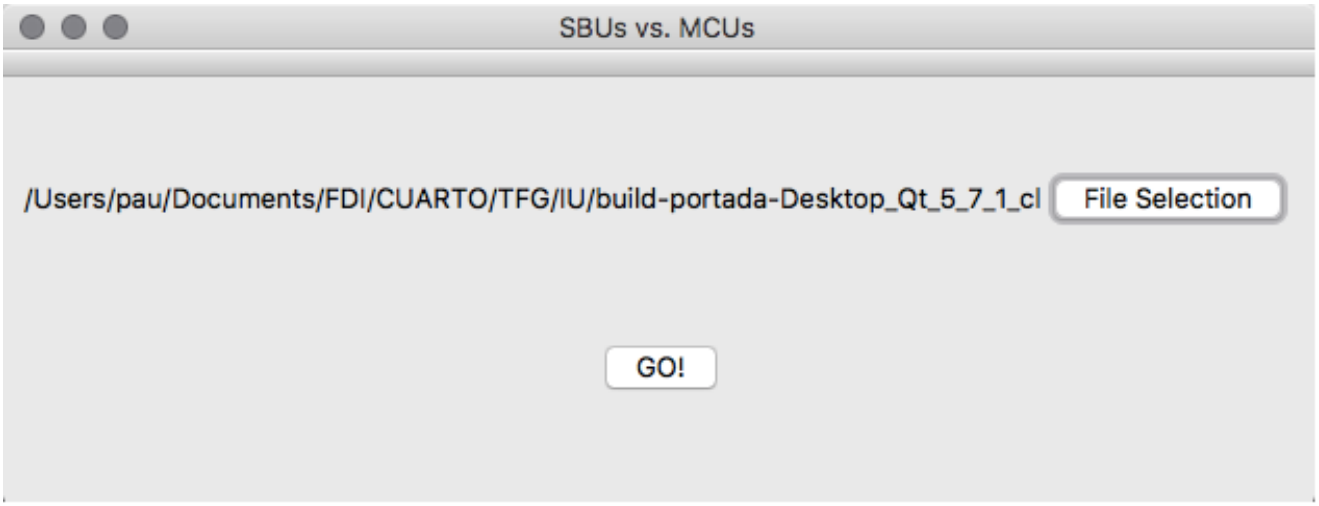

Figura 4.3: El fichero elegido se ha cargado en el label.

Una vez seleccionado el archivo, se actualiza una etiqueta label que muestra el nombre y la ruta en la que se encuentra el archivo y éste aparecera en la primera ventana mostrada. A continuación, al pulsar el botón de comienzo se ejecutará el programa principal con el archivo seleccionado.

Después, aparecerá una ventana nueva que contiene un dibujo de la memoria SRAM repartido en filas y columnas y se marcan sobre ella los puntos donde aparecen MCUs y la cantidad de los bits que los forman.

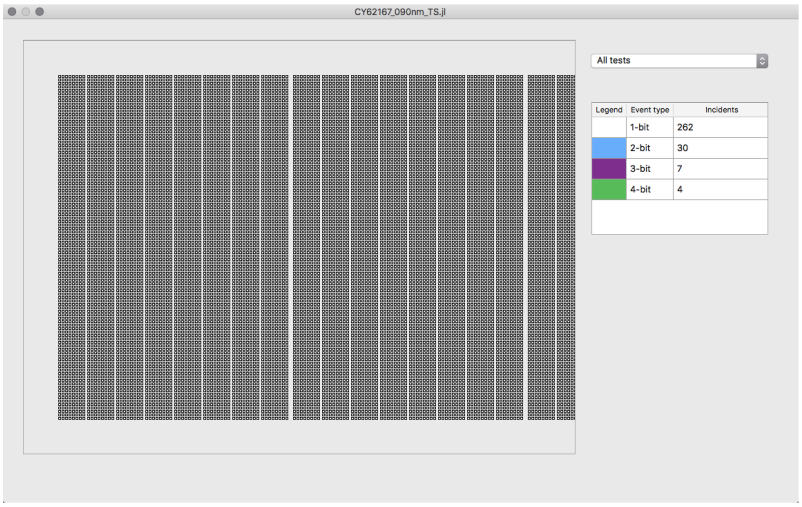

Figura 4.4: Visualización de las celdas de la memoria SRAM y localización de MCUs.

También se puede observar en la parte derecha de la ventana, una tabla que contiene una leyenda con los colores en los que aparecen los bits, los tipos de eventos que detectamos y el número de aparición de éstos.

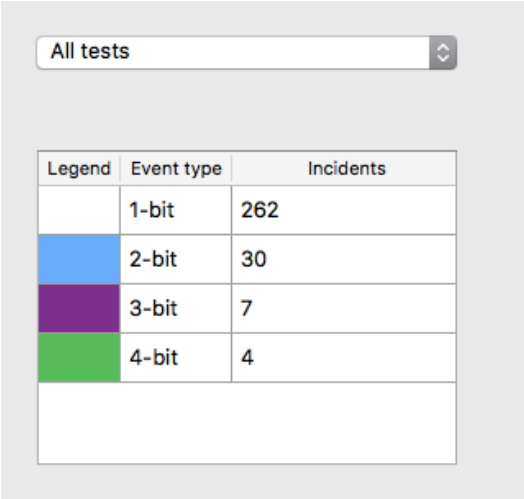

Figura 4.5: Visualización de la leyenda con los bits que aparecen, los distintos eventos y el número de aparición de éstos.

Finalmente, si queremos seleccionar un archivo distinto y ejecutar un experimento diferente, basta con volver a pulsar el botón de selección de archivo y elegir uno nuevo. El

programa se reiniciará y ejecutará el nuevo experimento generando de nuevo la ventana que contiene el dibujo de la memoria SRAM.

En el caso de que queramos salir de la aplicación, basta con pulsar en el icono superior que contiene un X para finalizar la ejecución.

Capítulo 5

Resultados

Todas las reglas representadas en el Capítulo 4 se han usado para elaborar una metodología que detecte MCUs. Esta metodología se explica en el Algoritmo 1. Si se conocen otros valores *DV* críticos (por ejemplo de experimentos con otro patrón) estos se deben incluir en los conjuntos *Candidates* y *PCandidates*.

Para verificar cuán efectiva es esta estrategia, se han usado los conjuntos de datos obtenidos al irradiar memorias SRAM de 90 y 130-nm con neutrones 14MeV en las instalaciones de GENEPI2. En estos experimentos, las memorias se escribieron con un patrón 0x55 y se irradiaron en diferentes rondas (Tablas 5.1 y 5.2). Los datos de la Tabla 5.1 no ha sido publicados aún, pero los de la Tabla 5.2 se han presentado y analizado en otro documento.

	SBU		2.bit MCU		3-bit MCU	
Test	Est.	Real	Est.	Real	Est.	Real
A	369	355	34	41	0	0
B	322	312	29	34	0	0
C	243	241	19	20	1	1
D	273	271	21	22	0	0
E	256	248	26	30	0	0
F	260	252	33	37	0	0

Tabla 5.1: 130-NM SRAM: Eventos estimados vs. eventos reales

	SBU		2.bit MCU		3-bit MCU		4-bit MCU	
Test	Est.	Real	Est.	Real	Est.	real	Est.	Real
A	1621	1645	112	96	14	12	8	8
B	1354	1385	105	89	11	10	2	3
C	1201	1215	105	96	11	13	5	3
D	1047	1065	109	97	14	15	5	4
E	879	876	95	99	15	12	3	4
F	727	734	93	77	11	16	4	5
G	623	623	70	69	5	7	1	0
	5-bit MCU		6-bit MCU		7-bit MCU		$\geq$ 8 bit	
A	0	2	1	0	0	0	0	1
B	1	0	0	1	0	0	0	0
C	0	1	0	0	0	0	0	0
D	0	0	0	0	0	1	0	0
E	0	0	0	1	0	0	0	0
F	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0

Tabla 5.2: 90-NM SRAM: Eventos estimados vs. eventos reales

Ambas tablas comparan el número estimado de MCUs con el número real. La distribución real de los MCUs se descubrió usando información privada del fabricante. En la Tabla 5.1 se ha subestimado el número de eventos. En otras palabras: el algoritmo no ha gestionado la identificación todos los valores anómalos. Sin embargo, en la Tabla 5.2, el caso es exactamente el contrario: se han sobreestimado los eventos múltiples. La razón fue que la posición exacta de las celdas depende de la posición del bit en la palabra. En este caso, algunos bitflips estaban lo suficientemente cerca como para engañar al algoritmo ya que las direcciones eran similares pero no pertenecían a los MCUs, porque las celdas que se trastornaban estaban muy lejos unas de otras.

Una posible mejora del algoritmo podría consistir en la incorporación de la posición del bit de la dirección afectada. Esto significa que, por ejemplo, palabras de 8-bits necesitarían 3 bits adicionales para codificar las celdas en lugar de todas las palabras. El problema es

que L_N se incrementa con un factor de 8, haciendo los cálculos más complejos.

También, el algoritmo falla cuando detecta eventos inusuales grandes. De todos modos, los eventos no detectados están en el orden de un 15 % del total, lo cual hace que sea una buena estimación teniendo en cuenta que no se requiere el conocimiento físico del diseño de la memoria y que este margen es del orden de error experimental (más o menos, el doble de la raíz cuadrada de los eventos). Finalmente, el tiempo de cómputo del programa que ha sido codificado por el Algoritmo 1 no es del todo muy alto. Así, la clasificación de todos los datos mostrados en la Tabla 5.2, solo requiere unos 45s en un PC con un procesador de 4 cores Intel Xeon, 8 Gb RAM, trabajando a 3.39 GHz, y usando Xubuntu GNU/Linux 16.04/64 bits.

En cuanto al tiempo de cómputo del programa hecho en Julia y la versión propuesta en lenguaje C, disminuye drasticamente cómo podemos observar en la Tabla 4.1. Dependiendo del tipo de memoria elegida obtenemos, en algunos casos, una mejora de más de la mitad del tiempo procesado poniendo en evidencia que el uso de lenguaje C es mucho más eficiente y rápido a la hora de procesar los datos que la versión de Julia. Esta reducción del tiempo de procesado es propiciado por la disminución del uso de memoria.

Capítulo 6

Conclusiones

6.1. Conclusiones

En este documento hemos podido comparar y evaluar los resultados obtenidos de las propiedades estadísticas del conjunto de direcciones con bitflips, con las predicciones matemáticas de un modelo donde todos los eventos son aislados. A su vez, dichos resultados se muestran de una forma mucho más amigable a través de una interfaz gráfica.

Hemos podido detectar que al usar un lenguaje diferente, en nuestro caso C para el programa principal, hemos obtenido una mejora en cuanto a rendimiento, disminuyendo el tiempo de ejecución en más de la mitad.

La implementación de las funciones estadísticas no ha sido muy compleja, pero sí que requiere de ciertos conocimientos matemáticos para su comprensión.También hay que destacar la inclusión de la interfaz gráfica como elemento significativo y fundamental de este proyecto. Con ella, conseguimos que cualquier usuario pueda tener una visión mucho mas clara, sencilla y limpia de los resultados obtenidos del experimento y también mejora sustancialmente el método de selección del experimento a ejecutar.

En conclusión podemos asegurar que se han cumplido los objetivos establecidos al comienzo de este documento. Primero, se ha hecho un análisis de las diferentes técnicas existentes para detectar los diferentes errores. A continuación, se ha desarrollado la herramienta que

además de realizar su función específica, mejora en tiempos de procesado y espacio a su antecesora.

Y finalmente, se ha creado una interfaz gráfica para facilitar la interacción del usuario con la herramienta. Podemos concluir también que ha sido un acierto la modificación que se ha realizado sobre el programa como así se ve reflejado en la mejora de resultados expuestos a lo largo de este documento.

6.2. Aportación individual de los miembros del grupo al proyecto

A rasgos generales, el trabajo se ha desarrollado de manera cooperativa, organizando sesiones conjuntas para implementar juntos los distintos aspectos del proyecto.

Al comienzo del proyecto, planteamos la optimización del nuevo programa a partir del programa base. Pensamos y detallamos que pasos seguiríamos y que estructura debería tener nuestra herramienta. Posteriormente, David se documentó a fondo sobre el lenguaje Julia, realizando distintas modificaciones del código con el fin de conocer y entender su funcionamiento para realizar el análisis del código. Paula comenzó la traducción a C, creando el proyecto del programa y sus diferentes archivos, así como de las estructuras que ibamos a usar. Posteriormente, y después de varios meses de pruebas y modificaciones, ambos finalizarían el programa.
Paula, una vez terminada la traducción, realizó una serie de pruebas con los datos proporcionados sobre el nuevo programa para comprobar que las salidas resultantes eran las correctas en todos los experimentos que se nos proporcionaron.Con el programa base finalizado sería el momento de programar la interfaz gráfica. Está hecha con el framework Qt, que permite

primero crear una vista de los elementos a añadir en el programa y después permite el envío de señales entre los distintos componentes para realizar acciones. David comenzó a investigar las distintas funcionalidades del framework, su organización en archivos y realizó ciertos programas de prueba. Mientras tanto, Paula comenzó con la redacción de la memoria del proyecto, analizando los requisitos necesarios y realizando un esqueleto del documento.

En la última parte del proyecto, debido a las dificultades encontradas a la hora de realizar la interfaz, Paula tuvo un importante peso en la realización y conclusión de la GUI, mientras que David continuaba y finalizaba la memoria.

En el transcurso del tiempo que se ha realizado este proyecto, ambos miembros del grupo se reunían de manera constante para realizar el trabajo en conjunto en un despacho facilitado por la directora del proyecto favoreciendo el trabajo en equipo, prestándonos ayuda entre nosotros y permitiendo una mejor comunicación.

Apéndice A

Código del proyecto

Este proyecto consta de tres archivos que contienen el código de nuestro programa.El primero contiene el código principal, denominado *main.c*, un segundo con la librería de funciones *libs.h* y otro con las estructuras que contienen los distintos tipos de datos que vamos a usar.

A continuación se adjunta el código en lenguaje C con las funciones y bibliotecas mas importantes que se han utilizado y explicado en este proyecto.

```
1  //** Librerías que hemos usado**//
2
3  #define libs_h
4  #include <stdio.h>
5  #include <string.h>
6  #include <inttypes.h>
7  #include <stdlib.h>
8  #include <float.h>
9  #include <math.h>
10 #include <stdbool.h>
11 #include <ctype.h>
12 #include <time.h>
13 #include <fcntl.h>
14 #include <math.h>
15
```

```
16
17
18  //** Principales Funciones **//
19
20
21
22  /*
23   * This function allows calculating the lowest number of repetitions from which
24   * the expected number is below threshold.
25   * RepInHistVector is a column vector containing the statistics.It is assumed
26   * that the first element correspond to 0 repetitions.
27   */
28  int32_t excessiveRepetitions(matrixInt322DStruct* repInHistVector, int repInHistVector
29  long int LN, char* operation, double threshold){
30          int32_t nthreshold = 0, ndv = 0, k, i;
31      vectorInt32Struct occurrenceIndex;
32      occurrenceIndex.length = repInHistVector->rows;
33          occurrenceIndex.data = calloc(occurrenceIndex.length, sizeof(int32_t));
34          for (i = 0; i < repInHistVector->rows; i++)
35                  occurrenceIndex.data[i] = i;
36
37          for (i = 0; i < occurrenceIndex.length; i++)
38                  ndv += ((int32_t)repInHistVector->data[i]
39          [repInHistVectorCol] * occurrenceIndex.data[i]);
40
41          k = 0;
42          if (strcmp(operation, "pos") == 0){
43          if ((excessiveRepetitions(repInHistVector, repInHistVectorCol, LN, "xor",
44          threshold) - 1) > 0){
45                  k = excessiveRepetitions(repInHistVector,
46          repInHistVectorCol, LN, "xor", threshold) - 1;
47          } else {
48              k = 0;
49          }
```

```
50      } else {
51          k = 0;
52      }
53
54      for (i = k; i < occurrenceIndex.length; i++) {
55          if (expectedRepetitions(i, LN, ndv, operation) < threshold) {
56              nthreshold = i;
57              break;
58          }
59      }
60      free(occurrenceIndex.data);
61          return nthreshold;
62  }
63
64
65
66  /*
67   * Keep in mind that davmatrix is a boolean matrix containing FALSE if two elements
68   * are not involved in the same MCU and TRUE if so. this function
69   *locate the elements and agrupate it in pairs
70   */
71  matrixInt322DStruct agrupate_mcus(matrixBool2DStruct* davmatrix){
72      /* Let us locate the elements */
73      /* Find the indexes with nonzero values */
74      int count = 0, i, j, k, x, xx;
75      vectorInt32Struct nonzerovectorelements;
76      nonzerovectorelements.length = davmatrix->rows * davmatrix->cols;
77      nonzerovectorelements.data = calloc(davmatrix->rows*davmatrix->cols,
78      sizeof(int32_t));
79      for (i = 0; i < davmatrix->rows; i++) {
80          for (j = 0; j < davmatrix->cols; j++) {
81              if (davmatrix->data[i][j] != false) {
82                  nonzerovectorelements.data[count] = j * davmatrix->cols + i;
83                  count++; /* Number of elements known but they can be repeated */
```

```
84              }
85          }
86      }
87      nonzerovectorelements.length = count;
88      nonzerovectorelements.data = realloc(nonzerovectorelements.data,
89      nonzerovectorelements.length * sizeof(int32_t));
90
91      /* Now, transform the elements in pairs creating a matrix with 2 columns. */
92      matrixInt322DStruct relatedpairs;
93      relatedpairs.rows = nonzerovectorelements.length;
94      relatedpairs.cols = 2;
95      relatedpairs.data = calloc(relatedpairs.rows, sizeof(int32_t*));
96      for (i = 0; i < relatedpairs.rows; i++) {
97          relatedpairs.data[i] = calloc(relatedpairs.cols, sizeof(int32_t));
98      }
99
100     /*
101      * The first operation calculates the column
102      */
103     for (i = 0; i < relatedpairs.rows; i++) {
104         if((nonzerovectorelements.data[i] % davmatrix->rows != 0)){
105         relatedpairs.data[i][0] = nonzerovectorelements.data[i]
106         }else{
107             relatedpairs.data[i][0] = nonzerovectorelements.data[i]
108         }
109         relatedpairs.data[i][1] = (nonzerovectorelements.data[i] /
110         davmatrix->rows)+1;
111     }
112
113     free(nonzerovectorelements.data);
114     /* We have created a matrix containing the related pairs. Now, we must
115      * group them in larger events.
116      * First step: A matrix initilizing the events. Perhaps *too large, but it is bett
117      * to have spare space.
```

```
118         */
119     matrixInt322DStruct thesummary;
120     thesummary.rows = count;
121     thesummary.data = calloc(count, sizeof(int32_t*));
122     if (count < NMaxAddressesInEvent) { /* If the minimun value is newCount */
123         thesummary.cols = count;
124         for (i = 0; i < count; i++) {
125             thesummary.data[i] = calloc(thesummary.cols, sizeof(int32_t));
126         }
127     } else { /* If the minimun value is NMaxAddressesInEvent */
128         thesummary.cols = NMaxAddressesInEvent;
129         for (i = 0; i < count; i++) {
130             thesummary.data[i] = calloc(thesummary.cols, sizeof(int32_t));
131         }
132     }
133
134     /* Now, we save the first pair in the first column: */
135     int totalEvents = 0, thelargestMCU = 0;
136     int32_t firstAddress, secondAddress, firstAddressRow = 0, secondAddressRow = 0;
137     for (k = 0; k < count; k++) {
138         /* There are several cases. Let us list them in order.
139          * Positions are extrated from the pairs. */
140         firstAddress = relatedpairs.data[k][0];
141         secondAddress = relatedpairs.data[k][1];
142         /* We need to look if firstAddress and secondAddress are in the matrix */
143         bool firstAddressFound = false, secondAddressFound = false;
144         for (i = 0; i < thesummary.rows; i++) {
145             for (j = 0; j < thesummary.cols; j++) {
146                 if ((thesummary.data[i][j] == firstAddress) &&
147                 (firstAddressFound == false)) {
148                     firstAddressFound = true;
149                     firstAddressRow = i;
150                 }
151                 if ((thesummary.data[i][j] == secondAddress)
```

```
152                 && (secondAddressFound == false)) {
153                     secondAddressFound = true;
154                     secondAddressRow = i;
155                 }
156             }
157         }
158         /*
159          * Case 1:
160          * Both addresses are not present in thesummary. They are placed at a new_row
161          * at NTotalEvents.
162          */
163         if (!firstAddressFound && !secondAddressFound){
164             thesummary.data[totalEvents][0] = relatedpairs.data[k][0];
165             thesummary.data[totalEvents][1] = relatedpairs.data[k][1];
166             totalEvents++;
167         }
168         /*
169          * Case 2:
170          * FirstAddress is present, but not SecondAddress
171          */
172         if (firstAddressFound && !secondAddressFound){
173             /* First of all, let us locate the row where the FirstAddress is.
174              * Now, it is necessary to locate the first free column to put the new
175              address. */
176             x = 0;
177             while ((thesummary.data[firstAddressRow][x] != 0)
178             && (x < thesummary.cols)) {
179                 x++;
180             }
181             /* Fixed problem. We do know exactly the position to put the new value. */
182             if (x < thesummary.cols) {
183                 thesummary.data[firstAddressRow][x] = secondAddress;
184             }
185         }
```

```
186         /*
187          * Case 3:
188          * FirstAddress absent, Second Address present. similar to previous one.
189          */
190         if (!firstAddressFound && secondAddressFound) {
191             /* First of all, let us locate the row where the Second Address is.
192              * Now, it is necessary to locate the first free
193              column to put the new address. */
194             x = 0;
195             while ((thesummary.data[secondAddressRow][x] != 0)
196             && (x < thesummary.cols)) {
197                 x++;
198             }
199             /* Fixed problem. We do know exactly the position to put the new value.
200             if (x < thesummary.cols) {
201                 thesummary.data[secondAddressRow][x] = firstAddress;
202             }
203         }
204         /*
205          * Case 4:
206          * Both addresses had been previously detected. Two subcases appear: They
207          * are included in the same row (MCU) so the program must skip this pair or
208          * are included in differen rows. Therefore, both rows
209          must be carefully merged.
210          */
211         if (firstAddressFound && secondAddressFound) {
212             /*
213              * If both values are identical, the pair must be skipped and the progra
214              * continue. If different, the rows must be merged.
215              */
216             if (firstAddressRow != secondAddressRow){
217                 /* First step: It is necessary to range the
218                 numbers in increasing order. */
219                 int32_t addressRowMin = 0, addressRowMax = 0;
```

```
220                 if (firstAddressRow < secondAddressRow) {
221                     addressRowMin = firstAddressRow;
222                     addressRowMax = secondAddressRow;
223                 } else {
224                     addressRowMin = secondAddressRow;
225                     addressRowMax = firstAddressRow;
226                 }
227 
228                 /* Now, let us locate elements in RowMin different from 0 */
229                 vectorInt32Struct thesummarytemp;
230                 thesummarytemp = unionAgrupate_mcus(&thesummary, addressRowMin,
231                 addressRowMax);
232 
233                 /*
234                  * Time to put the elements in the row.
235                  */
236                 for (i = 0; i < thesummary.cols; i++) {
237                     thesummary.data[addressRowMin][i] = thesummarytemp.data[i];
238                     /* The union vector is copied into the matrix */
239                     thesummary.data[addressRowMax][i] = 0;
240                 }
241                 /* Also, as two events have been merged, the
242                 number of total events is lower: */
243                 totalEvents--;
244             }
245         }
246     }
247 
248     liberaInt(relatedpairs.data, count);
249     /*
250      * Now, we will separate the cells with information from those with zeros.
251      * The number of rows is easy to calculate: NTotalEvents. Concerning the other
252      element:
253      */
```

```
254     int ind = 0, sum = 0;
255     for(j = 2; j < thesummary.cols; j++) {
256         for (i = 0; i < thesummary.rows; i++) {
257             if(thesummary.data[i][j] != 0){
258                 if(j > ind){
259                     ind = j;
260                 }
261                 sum++;
262             }
263         }
264         if (sum == 0) {
265             thelargestMCU = j;
266             break;
267         } else {
268             thelargestMCU = ind+1;
269         }
270     }
271     matrixInt322DStruct result;
272     result.rows = totalEvents;
273     result.cols = thelargestMCU;
274     result.data = calloc(result.rows, sizeof(int32_t*));
275     for (i = 0; i < result.rows; i++) {
276         result.data[i] = calloc(result.cols, sizeof(int32_t));
277     }
278     for (x = 0; x < result.rows; x++) {
279         for (xx = 0; xx < result.cols; xx++) {
280             result.data[x][xx] = thesummary.data[x][xx];
281         }
282     }
283 
284     liberaInt(thesummary.data, thesummary.rows);
285     return result;
286 }
287 
```

```
288 /*
289  * An alternative implementation of the trace rule.
290  * Anomal XOR values is used to avoid the repetition of elements.
291  */
292 vectorInt32Struct traceRule(matrixInt322DStruct *xorDVtotalrepetitions,
293 matrixUint322DStruct *totalDVhistogram, matrixInt322DStruct *anomalXORvalues,
294 long int LN){
295     int i, j, z, index;
296     int32_t nBits = (log(LN + 1) / log(2));
297 
298         /* Elements with 1-trace */
299         printf("\n\tInvestigating Trace = 1: ");
300     /* An array with nBits is created with 2^x values */
301     vectorInt32Struct candidatesT1;
302     candidatesT1.length = nBits;
303         candidatesT1.data = calloc(candidatesT1.length, sizeof(int32_t));
304         for (i = 0; i < candidatesT1.length; i++){
305                 candidatesT1.data[i] = pow(2, i);
306         }
307 
308         int32_t nT1Threshold = excessiveRepetitions(xorDVtotalrepetitions,
309     1, LN, "xor", randomnessThreshold);
310 
311     vectorInt32Struct selectedT1;
312     selectedT1.length = nBits;
313     selectedT1.data = calloc(selectedT1.length, sizeof(int32_t));
314     int sT1Lenght = 0;
315     for (i = 0; i < nBits; i++) {
316         if (totalDVhistogram->data[candidatesT1.data[i]-1][1] >= nT1Threshold) {
317             selectedT1.data[sT1Lenght] = candidatesT1.data[i];
318             sT1Lenght++;
319         }
320     }
321     selectedT1.length = sT1Lenght;
```

```
322     selectedT1.data = realloc(selectedT1.data, selectedT1.length * sizeof(int32_t));
323     free(candidatesT1.data);
324
325     vectorInt32Struct winnersT1;
326     winnersT1 = setdiffTraceRule(&selectedT1, anomalXORvalues);
327
328     free(selectedT1.data);
329     printf("%d candidate", winnersT1.length);
330     if (winnersT1.length != 1)
331         printf("s");
332     printf(" found.");
333
334     /* Elements with 2-trace */
335     printf("\n\tInvestigating Trace = 2: ");
336     vectorInt32Struct candidatesT2;
337     candidatesT2.length = 0.5 * nBits * (nBits-1);
338     candidatesT2.data = calloc(candidatesT2.length, sizeof(int32_t));
339     index = 0;
340     for (i = 0; i < (nBits-1); i++) {
341         for (j = i+1; j < nBits; j++) {
342             candidatesT2.data[index] = (pow(2, i))+(pow(2, j)); /* 2^k1 + 2^k2 */
343             index++;
344         }
345     }
346
347     int32_t nT2Threshold = excessiveRepetitions(xorDVtotalrepetitions, 1, LN, "xor",
348     randomnessThreshold);
349
350     vectorInt32Struct selectedT2;
351     selectedT2.length = candidatesT2.length;
352     selectedT2.data = calloc(candidatesT2.length, sizeof(int32_t));
353     int sT2Lenght = 0;
354     for (i = 0; i < candidatesT2.length; i++) {
355         if (totalDVhistogram->data[candidatesT2.data[i]-1][1] >= nT2Threshold) {
```

```
356             selectedT2.data[sT2Lenght] = candidatesT2.data[i];
357             sT2Lenght++;
358         }
359     }
360     selectedT2.length = sT2Lenght;
361     selectedT2.data = realloc(selectedT2.data, sT2Lenght * sizeof(int32_t));
362     free(candidatesT2.data);
363 
364     vectorInt32Struct winnersT2;
365     winnersT2 = setdiffTraceRule(&selectedT2, anomalXORvalues);
366 
367     free(selectedT2.data);
368     printf("%d candidate", winnersT2.length);
369     if (winnersT2.length != 1)
370         printf("s");
371     printf(" found.");
372 
373     /* Elements with 3-trace */
374     printf("\n\tInvestigating Trace = 3: ");
375     vectorInt32Struct candidatesT3;
376     candidatesT3.length = nBits*(nBits-1)*(nBits-2)/6;
377     candidatesT3.data = calloc(candidatesT3.length, sizeof(int32_t));
378     index = 0;
379     for (i = 0; i < (nBits-2); i++) {
380         for (j = i+1; j < (nBits-1); j++) {
381             for (z = j+1; z < nBits; z++) {
382                 candidatesT3.data[index] = (pow(2, i))+(pow(2, j))+(pow(2, z));
383                 index++;
384             }
385         }
386     }
387     vectorInt32Struct selectedT3;
388     selectedT3.data = calloc(candidatesT3.length, sizeof(int32_t));
389     int sT3Lenght = 0;
```

```
390     for (i = 0; i < candidatesT3.length; i++) {
391         if (totalDVhistogram->data[candidatesT3.data[i]-1][1] >= nT2Threshold) {
392             selectedT3.data[sT3Lenght] = candidatesT3.data[i];
393             sT3Lenght++;
394         }
395     }
396     selectedT3.length = sT3Lenght;
397     selectedT3.data = realloc(selectedT3.data, selectedT3.length * sizeof(int32_t));
398 
399     free(candidatesT3.data);
400 
401     vectorInt32Struct winnersT3;
402     winnersT3 = setdiffTraceRule(&selectedT3, anomalXORvalues);
403     printf("%d candidate", winnersT3.length);
404     free(selectedT3.data);
405     if (winnersT3.length != 1)
406         printf("s");
407     printf(" found.");
408     printf("\n\tEnd of search.\n");
409 
410     vectorInt32Struct vectorUnion = unionVec(&winnersT1, &winnersT2, &winnersT3);
411     free(winnersT1.data);
412     free(winnersT2.data);
413     free(winnersT3.data);
414 
415     return vectorUnion;
416 }
417 
418 /*
419  * Function to diferentiate where the MCUs values are.
420  */
421 matrixInt323DStruct propose_MCUs(char* op, matrixUint323DStruct *XORDVmatrix3D,
422 matrixUint323DStruct *POSDVmatrix3D, matrixInt322DStruct
423 *anomalXOR, matrixInt322DStruct *anomalPOS, vectorIntStruct*
```

```
424 nAddressesInRound){
425         int i, j, ktest;
426         int rows = ceil(XORDVmatrix3D->rows / 2 - 1);
427         if (strcmp(op, "xor") == 0){ /* OP == XOR */
428         matrixInt323DStruct XOR_summary;
429         XOR_summary.rows = rows;
430         XOR_summary.cols = UnrealMCUsize;
431         XOR_summary.dims = XORDVmatrix3D->dims;
432         XOR_summary.data = calloc(XOR_summary.rows, sizeof(int32_t**));
433                 for (i = 0; i < XOR_summary.rows; i++) {
434                         XOR_summary.data[i] = calloc(XOR_summary.cols,
435             sizeof(int32_t*));
436                         for (j = 0; j < XOR_summary.cols; j++) {
437                                 XOR_summary.data[i][j] = calloc(XOR_summary.dims,
438                 sizeof(int32_t));
439                         }
440                 }
441                 for (ktest = 0; ktest < XOR_summary.dims; ktest++){
442             matrixUint322DStruct xordvmatrix;
443             xordvmatrix.rows = nAddressesInRound->data[ktest];
444             xordvmatrix.cols = nAddressesInRound->data[ktest];
445             xordvmatrix.data = calloc(xordvmatrix.rows, sizeof(int32_t*));
446                         for (i = 0; i < xordvmatrix.rows; i++) {
447                                 xordvmatrix.data[i] =
448                 calloc(xordvmatrix.cols, sizeof(int32_t));
449                         }
450             copyOfMatrix3to2(XORDVmatrix3D, &xordvmatrix, ktest);
451
452                         matrixBool2DStruct XORmarked_pairs =
453             marking_addresses(&xordvmatrix, anomalXOR);
454
455                         matrixInt322DStruct XORproposed_MCUs =
456             agrupate_mcus(&XORmarked_pairs);
457                         for (i = 0; i < XORproposed_MCUs.rows; i++) {
```

```
458                             for (j = 0; j < XORproposed_MCUs.cols; j++) {
459                                     XOR_summary.data[i][j][ktest] =
460                 XORproposed_MCUs.data[i][j];
461                             }
462                         }
463                 }
464
465         /* However, as there are too many zeros, it is
466         *interesting to reshape the matrix.
467          * Thus, some memory is saved.But we must be careful
468          *with the different dimensions
469          * of every partial XY matrix in XYZ arrays.A simple
470          *but unelegant way is : */
471         matrixInt323DStruct finalXOR_summary = cutZerosFromMCUsummary(&XOR_summary);
472         libera3D(XOR_summary.data, XOR_summary.rows, XOR_summary.cols);
473         return finalXOR_summary;
474         }
475         else if (strcmp(op, "pos") == 0){ /* OP == POS */
476         matrixInt323DStruct POS_summary;
477         POS_summary.rows = rows;
478         POS_summary.cols = UnrealMCUsize;
479         POS_summary.dims = XORDVmatrix3D->dims;
480         POS_summary.data = calloc(POS_summary.rows, sizeof(int32_t**));
481                 for (i = 0; i < POS_summary.rows; i++) {
482                         POS_summary.data[i] = calloc(POS_summary.cols, sizeof(int32_
483                         for (j = 0; j < POS_summary.cols; j++) {
484                                 POS_summary.data[i][j] =
485                 calloc(POS_summary.dims, sizeof(int32_t));
486                         }
487                 }
488
489                 for (ktest = 0; ktest < XORDVmatrix3D->dims; ktest++){
490             matrixInt322DStruct posdvmatrix;
491             posdvmatrix.rows = nAddressesInRound->data[ktest];
```

```
492             posdvmatrix.cols = nAddressesInRound->data[ktest];
493             posdvmatrix.data = calloc(posdvmatrix.rows, sizeof(int32_t*));
494                         for (i = 0; i < posdvmatrix.rows; i++) {
495                                 posdvmatrix.data[i] =
496                     calloc(posdvmatrix.cols, sizeof(int32_t));
497                         }
498             copyOfMatrix3to2(POSDVmatrix3D, &posdvmatrix, ktest);
499
500             matrixBool2DStruct POSmarked_pairs;
501             POSmarked_pairs = marking_addresses(&posdvmatrix, anomalPOS);
502
503                         /* The following results only concern an operation.Just presen
504                          * operations.In practical use, only CMB should be used. */
505                         matrixInt322DStruct POSproposed_MCUs = agrupate_mcus(&POSmarke
506                         for (i = 0; i < POSproposed_MCUs.rows; i++) {
507                                 for (j = 0; j < POSproposed_MCUs.rows; j++) {
508                                         POS_summary.data[i][j][ktest] =
509                     POSproposed_MCUs.data[i][j];
510                                 }
511                         }
512                 }
513             /* However, as there are too many zeros, it is
514             *interesting to reshape the matrix.
515              * Thus, some memory is saved.But we must be careful
516              *with the different dimensions
517              * of every partial XY matrix in XYZ arrays.A simple
518              *but unelegant way is : */
519             matrixInt323DStruct finalPOS_summary = cutZerosFromMCUsummary(&POS_summary);
520             libera3D(POS_summary.data, POS_summary.rows, POS_summary.cols);
521             return finalPOS_summary;
522             }
523             else{ /* OP == CMB */
524             matrixInt323DStruct CMB_summary;
525             CMB_summary.rows = rows;
```

```
526         CMB_summary.cols = UnrealMCUsize;
527         CMB_summary.dims = XORDVmatrix3D->dims;
528         CMB_summary.data = calloc(CMB_summary.rows, sizeof(int32_t**));
529                 for (i = 0; i < CMB_summary.rows; i++) {
530                         CMB_summary.data[i] =
531             calloc(CMB_summary.cols, sizeof(int32_t*));
532                         for (j = 0; j < CMB_summary.cols; j++) {
533                                 CMB_summary.data[i][j] = calloc(CMB_summary.dims, si
534                         }
535                 }
536
537                 for (ktest = 0; ktest < XORDVmatrix3D->dims; ktest++){
538             matrixInt322DStruct xordvmatrix;
539             xordvmatrix.rows = nAddressesInRound->data[ktest];
540             xordvmatrix.cols = nAddressesInRound->data[ktest];
541             xordvmatrix.data = calloc(xordvmatrix.rows, sizeof(int32_t*));
542                         for (i = 0; i < xordvmatrix.rows; i++) {
543                                 xordvmatrix.data[i] =
544                 calloc(xordvmatrix.cols, sizeof(int32_t));
545                         }
546             copyOfMatrix3to2(XORDVmatrix3D, &xordvmatrix, ktest);
547
548             matrixInt322DStruct posdvmatrix;
549             posdvmatrix.rows = nAddressesInRound->data[ktest];
550             posdvmatrix.cols = nAddressesInRound->data[ktest];
551             posdvmatrix.data = calloc(posdvmatrix.rows, sizeof(int32_t*));
552                         for (i = 0; i < posdvmatrix.rows; i++) {
553                                 posdvmatrix.data[i] =
554                 calloc(posdvmatrix.cols, sizeof(int32_t));
555
556                         }
557             copyOfMatrix3to2(POSDVmatrix3D, &posdvmatrix, ktest);
558
559             matrixBool2DStruct XORmarked_pairs =
```

```
560             marking_addresses(&xordvmatrix, anomalXOR);
561                             matrixBool2DStruct POSmarked_pairs =
562             marking_addresses(&posdvmatrix, anomalPOS);
563
564             matrixBool2DStruct XOR_POS_marked_pairs;
565             XOR_POS_marked_pairs.rows =
566             nAddressesInRound->data[ktest];
567             XOR_POS_marked_pairs.cols =
568             nAddressesInRound->data[ktest];
569             XOR_POS_marked_pairs.data =
570             calloc(nAddressesInRound->data[ktest], sizeof(bool*));
571                             for (i = 0; i < nAddressesInRound->data[ktest]; i++) {
572                                     XOR_POS_marked_pairs.data[i] = calloc(nAddressesInRound
573                             }
574
575                             for (i = 0; i < nAddressesInRound->data[ktest]; i++) {
576                                     for (j = 0; j < nAddressesInRound->data[ktest]; j++) {
577                                             XOR_POS_marked_pairs.data[i][j] = XORmarked_pa
578                                     }
579                             }
580
581                             matrixInt322DStruct CMBproposed_MCUs =
582             agrupate_mcus(&XOR_POS_marked_pairs);
583                             for (i = 0; i < CMBproposed_MCUs.rows; i++) {
584                                     for (j = 0; j < CMBproposed_MCUs.cols; j++) {
585                                             CMB_summary.data[i][j][ktest] =
586                         CMBproposed_MCUs.data[i][j];
587                     }
588                             }
589                     }
590             /* However, as there are too many zeros, it is
591             *interesting to reshape the matrix.
592              * Thus, some memory is saved.But we must be careful
593              *with the different dimensions
```

```
594             * of every partial XY matrix in XYZ arrays.A simple
595             *but unelegant way is : */
596             matrixInt323DStruct finalCMB_summary;
597             finalCMB_summary = cutZerosFromMCUsummary(&CMB_summary);
598             libera3D(CMB_summary.data, CMB_summary.rows, CMB_summary.cols);
599             return finalCMB_summary;
600             }
601 }
602
603
604 /*
605 *this function avoids the selection of false MCU indicators.
606 */
607 matrixInt322DStruct extractAnomalDVSelfConsistency(char* op,
608 matrixInt322DStruct* opDVtotalrepetitions,
609 matrixUint322DStruct* totalDVhistogram,
610 vectorIntStruct* nAddressesInRound,
611 matrixInt322DStruct* opdvhistogram,
612 matrixInt323DStruct* opdvmatrixbackup, int* XORANOMALS){
613             int32_t opNthreshold;
614             int* n_anomalous_values = malloc(sizeof(int));
615     matrixInt322DStruct testOPa;
616     testOPa.rows = 0;
617     testOPa.cols = 0;
618             matrixInt322DStruct oPExtracted_values;
619             bool oPSelfConsistence = true;
620             int n_anomalous_repetitions = 1, i, j, test;
621             printf("\n\tDetermining the threshold for repetition excess:\n");
622
623     if (strcmp(op, "xor") == 0) {
624                     printf("\n\t\tXOR operation...");
625                     opNthreshold = excessiveRepetitions(opDVtotalrepetitions, 1,
626             totalDVhistogram->rows, "xor", randomnessThreshold);
627             testOPa = find_anomalies_histogram(totalDVhistogram, 1,
```

```
628         opDVtotalrepetitions,
629         1, opNthreshold, n_anomalous_values);
630         *XORANOMALS = *n_anomalous_values;
631 
632                 printf("\n\tXOR operation:\n");
633                 oPExtracted_values =
634         extract_some_critical_values(&testOPa, n_anomalous_values, 1);
635 
636                 while (oPSelfConsistence) {
637                         printf("\t\tStep %d ", n_anomalous_repetitions);
638                         oPExtracted_values =
639             extract_some_critical_values(&testOPa,
640             n_anomalous_values, n_anomalous_repetitions);
641 
642             matrixInt322DStruct opPartRepsSelfCons;
643             opPartRepsSelfCons.rows = oPExtracted_values.rows;
644             opPartRepsSelfCons.cols = nAddressesInRound-
645             >length;
646             opPartRepsSelfCons.data = calloc(opPartRepsSelfCons.rows,
647             sizeof(int32_t*));
648                         for (i = 0; i < opPartRepsSelfCons.rows; i++) {
649                                 opPartRepsSelfCons.data[i] =
650                     calloc(opPartRepsSelfCons.cols, sizeof(int32_t));
651                         }
652                         for (i = 0; i < opPartRepsSelfCons.rows; i++) {
653                     for (j = 0; j < opPartRepsSelfCons.cols; j++) {
654                                         opPartRepsSelfCons.data[i][j] =
655                         opdvhistogram->data[(oPExtracted_values.data[i][0])-1][j+1];
656                                 }
657                         }
658 
659                         for (test = 0; test < nAddressesInRound->length; test++) {
660                                 printf("Test %d", test+1);
661                     matrixUint322DStruct opdvmatrix;
```

```
662                 opdvmatrix.rows = nAddressesInRound->data[test];
663                 opdvmatrix.cols = nAddressesInRound->data[test];
664                 opdvmatrix.data = calloc(opdvmatrix.rows, sizeof(int32_t*));
665                                 for (i = 0; i < opdvmatrix.rows; i++) {
666                                         opdvmatrix.data[i] =
667             calloc(opdvmatrix.cols, sizeof(int32_t));
668                                 }
669                 copyOfMatrix3to2(opdvmatrixbackup, &opdvmatrix, test);
670                 matrixBool2DStruct opMarkedPairs;
671                 opMarkedPairs = marking_addresses(&opdvmatrix, &oPExtracted_values);
672
673                 matrixInt322DStruct opProposedMcus;
674                 opProposedMcus = agrupate_mcus(&opMarkedPairs);
675                 int largestMCUSize = opProposedMcus.cols;
676
677                 int continuation = findEqual(&opPartRepsSelfCons, 0,
678                 test, largestMCUSize);
679                 if (continuation != 0) {
680                     oPSelfConsistence = false;
681                 }
682
683                         }
684                         if (oPSelfConsistence) {
685                                 n_anomalous_repetitions++;
686                                 printf("\n");
687                                 if (n_anomalous_repetitions > *n_anomalous_values) {
688                                         oPSelfConsistence = false;
689                                         printf("\n\t\tNo more anomalous elements to
690             check. Exiting\n");
691                                 }
692         }
693         else {
694                 printf("\n\t\tViolation of self-consistence. Returning to previous s
695                 n_anomalous_repetitions--;
```

```
696                 oPExtracted_values = extract_some_critical_values(&testOPa,
697                 n_anomalous_values, n_anomalous_repetitions);
698 
699             }
700         }
701 
702         } else if (strcmp(op, "pos") == 0){
703                 printf("\n\t\tPOS operation (It can take a long if there are too many a
704                 opNthreshold = excessiveRepetitions(opDVtotalrepetitions, 1,
705         totalDVhistogram->rows, "pos", randomnessThreshold);
706         testOPa = find_anomalies_histogram(totalDVhistogram, 2,
707         opDVtotalrepetitions, 1, opNthreshold, n_anomalous_values);
708 
709                 printf("\n\tPOS operation:\n");
710         oPExtracted_values =
711         xtract_some_critical_values(&testOPa, n_anomalous_values, 1);
712 
713         while (oPSelfConsistence) {
714                         printf("\t\tStep %d ", n_anomalous_repetitions);
715             oPExtracted_values = extract_some_critical_values(&testOPa, n_anomalous_val
716 
717             matrixInt322DStruct opPartRepsSelfCons;
718             opPartRepsSelfCons.rows = oPExtracted_values.rows;
719             opPartRepsSelfCons.cols = nAddressesInRound->length;
720             opPartRepsSelfCons.data = calloc(opPartRepsSelfCons.rows, sizeof(int32_t*))
721             for (i = 0; i < opPartRepsSelfCons.rows; i++) {
722                 opPartRepsSelfCons.data[i] =
723                 calloc(opPartRepsSelfCons.cols, sizeof(int32_t));
724             }
725             for (i = 0; i < opPartRepsSelfCons.rows; i++) {
726                 for (j = 0; j < opPartRepsSelfCons.cols; j++) {
727                     opPartRepsSelfCons.data[i][j] =
728                     opdvhistogram->data[(oPExtracted_values.data[i][0])-1][j+1];
729                 }
```

```
730             }
731
732                     for (test = 0; test < nAddressesInRound->length; test++) {
733                             printf("Test %d", test);
734
735                 matrixUint322DStruct opdvmatrix;
736                 opdvmatrix.rows = nAddressesInRound->data[test];
737                 opdvmatrix.cols = nAddressesInRound->data[test];
738                 opdvmatrix.data = calloc(opdvmatrix.rows, sizeof(int32_t*));
739                 for (i = 0; i < opdvmatrix.rows; i++) {
740                     opdvmatrix.data[i] = calloc(opdvmatrix.cols, sizeof(int32_t));
741                 }
742                 copyOfMatrix3to2(opdvmatrixbackup, &opdvmatrix, test);
743                 matrixBool2DStruct opMarkedPairs =
744                 marking_addresses(&opdvmatrix, &oPExtracted_values);
745
746                 matrixInt322DStruct opProposedMcus = agrupate_mcus(&opMarkedPairs);
747                 int largestMCUSize = opProposedMcus.cols;
748
749                 int continuation = findEqual(&opPartRepsSelfCons, 0,
750                 test, largestMCUSize);
751                 if (continuation != 0) {
752                     oPSelfConsistence = false;
753                 }
754
755                         }
756                         if (oPSelfConsistence) {
757                 n_anomalous_repetitions++;
758                 printf("\n");
759                 if (n_anomalous_repetitions > *XORANOMALS) {
760                     oPSelfConsistence = false;
761                     printf("\n\t\tNo more anomalous elements to check. Exiting\n");
762                 }
763             }else {
```

```
764                 printf("\n\t\tViolation of self-consistence. Returning to previous stat
765                 n_anomalous_repetitions--;
766                 oPExtracted_values = extract_some_critical_values(&testOPa,
767                 n_anomalous_values, n_anomalous_repetitions);
768             }
769         }
770 
771         }
772         return oPExtracted_values;
773 }
774 
775 /*
776 *This function gets the critical values from the XOR operation
777 *
778 */
779 
780 vectorInt32Struct criticalXORValuesFromClusters(matrixInt323DStruct* PrelMCUSummary,
781 matrixInt322DStruct* AddressMatrix,
782 matrixInt322DStruct* XORextracted_values,
783 matrixInt322DStruct* xorDVtotalrepetitions,
784 matrixInt322DStruct* DVHistogram){
785     vectorInt32Struct newCandidates;
786     newCandidates.length = 0;
787     newCandidates.data = calloc(PrelMCUSummary->rows *
788     PrelMCUSummary->cols * PrelMCUSummary->dims, sizeof(int32_t));
789         int ktest, kRow, kAddress1, kAddress2, i;
790     int32_t index1, index2, address1, address2, candidate;
791         for (ktest = 0; ktest < PrelMCUSummary->dims; ktest++){
792                 for (kRow = 0; kRow < PrelMCUSummary->rows; kRow++){
793                         for (kAddress1 = 1; kAddress1 < PrelMCUSummary->cols;
794             Address1++){
795                                 index1 = PrelMCUSummary->data[kRow][kAddress1][ktest];
796                                 if (index1 == 0){
797                                         break;
```

```
798                                 }
799                                 else{
800                                         address1 = AddressMatrix->data[index1-1][kte
801                                         for (kAddress2 = 0; kAddress2 < kAddress1; k
802                                                 index2 = PrelMCUSummary->data[kRow][
803                                                 address2 = AddressMatrix->data[index
804                                                 candidate = address1 ^ address2;
805                                                 if (findEqualMatrix(XORextracted_val
806                         candidate) == 0){
807                             vectorInt32Struct vCandidate;
808                             vCandidate.length = 1;
809                             vCandidate.data = calloc(1, sizeof(int32_t));
810                             vCandidate.data[0] = candidate;
811                             newCandidates = unionVec(&newCandidates,
812                             &vCandidate, NULL);
813                             free(vCandidate.data);
814                                                 }
815                                         }
816                                 }
817                         }
818                 }
819     }
820     newCandidates.data = realloc(newCandidates.data,
821     newCandidates.length* sizeof(int32_t));
822
823         int32_t xorNthreshold = excessiveRepetitions(xorDVtotalrepetitions, 1,
824     DVHistogram->rows, "xor", randomnessThreshold);
825
826     vectorInt32Struct purgedCandidatesXOR;
827     purgedCandidatesXOR.length = newCandidates.length;
828     purgedCandidatesXOR.data = calloc(purgedCandidatesXOR.length, sizeof(int32_t));
829     int purgedCandidatesLenght = 0;
830     for (i = 0; i < purgedCandidatesXOR.length; i++) {
831         if (DVHistogram->data[newCandidates.data[i]-1][1] >= xorNthreshold) {
```

```
832             purgedCandidatesXOR.data[purgedCandidatesLenght] =
833             newCandidates.data[i];
834             purgedCandidatesLenght++;
835         }
836     }
837     purgedCandidatesXOR.length = purgedCandidatesLenght;
838     purgedCandidatesXOR.data = realloc(purgedCandidatesXOR.data,
839     purgedCandidatesXOR.length * sizeof(int32_t));
840 
841         return purgedCandidatesXOR;
842 }
843 
844 /*
845  * This function allows the get of all the anomal values from the summaries.
846  */
847 void extractAnomalDVfromClusters(matrixInt322DStruct* content,
848 matrixInt322DStruct* XORextracted_values,
849 matrixInt322DStruct* POSextracted_values,
850 matrixInt322DStruct* xorDVtotalrepetitions
851 , matrixInt322DStruct* posDVtotalrepetitions,
852 matrixUint322DStruct* totalDVhistogram,
853 matrixInt323DStruct* xordvmatrixbackup,
854 matrixInt323DStruct* posdvmatrixbackup,
855 vectorIntStruct* nAddressesInRound, long int LN,
856 vectorInt32Struct* discoveredXORDVs,
857 vectorInt32Struct* discoveredPOSDVs,
858 matrixInt322DStruct* tempXORDVvalues,
859 matrixInt322DStruct* tempPOSDVvalues){
860 
861     bool foundNewDVValues = true;
862     int step = 0, i, j;
863 
864     tempXORDVvalues->rows = XORextracted_values->rows;
865     tempXORDVvalues->cols = XORextracted_values->cols;
```

```
866     tempXORDVvalues->data = calloc(tempXORDVvalues->rows, sizeof(int32_t*));
867     for (i = 0; i < tempXORDVvalues->rows; i++) {
868         tempXORDVvalues->data[i] = calloc(tempXORDVvalues->cols, sizeof(int32_t));
869     }
870 
871     tempPOSDVvalues->rows = POSextracted_values->rows;
872     tempPOSDVvalues->cols = POSextracted_values->cols;
873     tempPOSDVvalues->data = calloc(tempPOSDVvalues->rows, sizeof(int32_t*));
874     for (i = 0; i < tempPOSDVvalues->rows; i++) {
875         tempPOSDVvalues->data[i] = calloc(tempPOSDVvalues->cols, sizeof(int32_t));
876     }
877 
878     for (i = 0; i < tempXORDVvalues->rows; i++) {
879         for (j = 0; j < tempXORDVvalues->cols; j++) {
880             tempXORDVvalues->data[i][j] = XORextracted_values->data[i][j];
881         }
882     }
883     for (i = 0; i < tempPOSDVvalues->rows; i++) {
884         for (j = 0; j < tempPOSDVvalues->cols; j++) {
885             tempPOSDVvalues->data[i][j] = POSextracted_values->data[i][j];
886         }
887     }
888 
889     while (foundNewDVValues) {
890         step++;
891         printf("\nStep: %d", step);
892         printf("Creating preliminary organization...");
893         matrixInt323DStruct tempCMB_summary;
894         matrixInt323DStruct tempXOR_summary;
895         matrixInt323DStruct tempPOS_summary;
896 
897         tempXOR_summary = propose_MCUs("xor", xordvmatrixbackup,
898         posdvmatrixbackup, tempXORDVvalues, tempPOSDVvalues, nAddressesInRound);
899         tempPOS_summary = propose_MCUs("pos", xordvmatrixbackup,
```

```
900         posdvmatrixbackup, tempXORDVvalues, tempPOSDVvalues, nAddressesInRound);
901         tempCMB_summary = propose_MCUs("cmb", xordvmatrixbackup,
902         posdvmatrixbackup, tempXORDVvalues, tempPOSDVvalues, nAddressesInRound);
903
904         printf(" Ended. Searching new Elements...");
905
906         printf(" XOR...");
907
908         int nProposedXORDV = 0;
909         vectorInt32Struct proposedXORDV;
910         proposedXORDV = criticalXORValuesFromClusters(&tempCMB_summary, content,
911         tempXORDVvalues, xorDVtotalrepetitions, totalDVhistogram);
912
913         vectorInt32Struct proposedPOSDV;
914         nProposedXORDV = proposedXORDV.length;
915         int nProposedPOSDV = 0;
916         printf(" POS. SUB...");
917
918         printf(" Ended. ");
919
920         if (nProposedXORDV != 0) {
921             printf("\n\t\tXOR operation:  %d new DV element", nProposedXORDV);
922             if (nProposedXORDV != 1) {
923                 printf("s");
924             }
925             printf("found.");
926             *discoveredXORDVs = unionVec(&proposedXORDV, discoveredXORDVs, NULL);
927
928             matrixInt322DStruct auxXORDV;
929             auxXORDV.rows = XORextracted_values->rows + nProposedXORDV;
930             auxXORDV.cols = XORextracted_values->cols;
931             auxXORDV.data = calloc(auxXORDV.rows, sizeof(int32_t*));
932             for (i = 0; i < auxXORDV.rows; i++) {
933                 auxXORDV.data[i] = calloc(auxXORDV.cols, sizeof(int32_t));
```

```
934             }
935             for (i = 0; i < XORextracted_values->rows; i++) {
936                 for (j = 0; j < XORextracted_values->cols; j++) {
937                     auxXORDV.data[i][j] = tempXORDVvalues->data[i][j];
938                 }
939             }
940             int index = XORextracted_values->rows;
941             for (int a = 0; a < nProposedXORDV; a++) {
942                 auxXORDV.data[index][0] = proposedXORDV.data[a];
943                 auxXORDV.data[index][1] = totalDVhistogram->data[proposedXORDV.data[
944                 index++;
945             }
946 
947             tempXORDVvalues->data = auxXORDV.data;
948 
949             tempXORDVvalues->rows = XORextracted_values->rows + nProposedXORDV;
950             XORextracted_values->rows = XORextracted_values->rows + nProposedXORDV;
951 
952         }
953         if (nProposedPOSDV != 0) {
954             printf("\n\t\tPOS operation:  %d new DV element", nProposedPOSDV);
955             if (nProposedPOSDV != 1) {
956                 printf("s");
957             }
958             printf("found.");
959             *discoveredPOSDVs = unionVec(&proposedPOSDV, discoveredPOSDVs, NULL);
960 
961             matrixInt322DStruct auxPOSDV;
962             auxPOSDV.rows = POSextracted_values->rows + nProposedPOSDV;
963             auxPOSDV.cols = POSextracted_values->cols;
964             auxPOSDV.data = calloc(auxPOSDV.rows, sizeof(int32_t*));
965             for (i = 0; i < auxPOSDV.rows; i++) {
966                 auxPOSDV.data[i] = calloc(auxPOSDV.cols, sizeof(int32_t));
967             }
```

```
968                 for (i = 0; i < auxPOSDV.rows; i++) {
969                     for (j = 0; j < auxPOSDV.cols; j++) {
970                         auxPOSDV.data[i][j] = tempPOSDVvalues->data[i][j];
971                     }
972                 }
973                 int index = POSextracted_values->rows;
974                 for (int a = 0; a < nProposedPOSDV; a++) {
975                     auxPOSDV.data[index][0] = proposedPOSDV.data[a];
976                     auxPOSDV.data[index][1] = totalDVhistogram->data[proposedPOSDV.data[a]
977                     index++;
978                 }
979 
980                 tempPOSDVvalues->data = auxPOSDV.data;
981                 tempXORDVvalues->rows = POSextracted_values->rows + nProposedPOSDV;
982                 POSextracted_values->rows = POSextracted_values->rows + nProposedPOSDV;
983 
984             }
985             if ((nProposedXORDV == 0) && (nProposedPOSDV == 0)) {
986                 foundNewDVValues = false;
987                 printf("\n\n\t\tNO MORE ELEMENTS DISCOVERED. Stopping analysis.");
988             }
989 
990         }
991     }
992 
993     /*
994      * This function gets the critical values from the XOR operation
995      * Returns a matrix with these values.
996      */
997     matrixInt322DStruct criticalXORvaluesFromXORingRule(matrixInt322DStruct*
998     XORextracted_values, matrixInt322DStruct*
999     XORDVtotalrepetitions, matrixUint322DStruct*
1000    totalDVhistogram, vectorInt32Struct* candidates){
1001            int i, j;
```

```
1002            bool candidateFind = false, prelCandidateFind = false;
1003            printf("\n\tCase 1: XORing known values... ");
1004        vectorInt32Struct prelCandidates;
1005        prelCandidates.data = calloc(1, sizeof(int32_t));
1006        prelCandidates.length = 0;
1007
1008        vectorInt32Struct oldXORValues;
1009        oldXORValues.length = XORextracted_values->rows;
1010        oldXORValues.data = calloc(oldXORValues.length, sizeof(int32_t));
1011        for (i = 0; i < oldXORValues.length; i++) {
1012            oldXORValues.data[i] = XORextracted_values->data[i][0];
1013        }
1014
1015        sort(oldXORValues);
1016
1017            int k1, k2;
1018            for (k1 = 0; k1 < XORextracted_values->rows; k1++){
1019                    int old1 = oldXORValues.data[k1];
1020                    for (k2 = k1 + 1; k2 < XORextracted_values->rows; k2++){
1021                            candidateFind = false, prelCandidateFind = false;
1022                            int old2 = oldXORValues.data[k2];
1023                            int candidate = old1^old2;
1024                if ((findEqualVector(&oldXORValues, candidate) == 0) &&
1025                (findEqualVector(&prelCandidates, candidate) == 0)) {
1026                    vectorInt32Struct vCandidate;
1027                    vCandidate.length = 1;
1028                    vCandidate.data = calloc(1, sizeof(int32_t));
1029                    vCandidate.data[0] = candidate;
1030                    prelCandidates = unionVec(&prelCandidates, &vCandidate, NULL);
1031                }
1032                    }
1033            }
1034            sort(prelCandidates);
1035
```

```
1036        int nThresholdCase1 = excessiveRepetitions(XORDVtotalrepetitions, 1, totalDVhis
1037
1038    vectorInt32Struct candidatesCase1;
1039    candidatesCase1.length = 0;
1040    candidatesCase1.data = calloc(prelCandidates.length, sizeof(int32_t)); /* Later, t
1041    for (i = 0; i < prelCandidates.length; i++) {
1042        if (totalDVhistogram->data[prelCandidates.data[i]-1][1] >= nThresholdCase1) {
1043            candidatesCase1.data[candidatesCase1.length] = prelCandidates.data[i];
1044            candidatesCase1.length++;
1045        }
1046    }
1047    candidatesCase1.data = realloc(candidatesCase1.data,
1048    candidatesCase1.length*sizeof(int32_t));
1049
1050        if (candidatesCase1.length > 0){
1051                printf("%d new value", candidatesCase1.length);
1052                if (candidatesCase1.length != 1)
1053                        printf("s");
1054                printf(" obtained. \n");
1055        }
1056        else{
1057                printf("No new values. Going on.\n");
1058                }
1059
1060    oldXORValues = unionVec(&oldXORValues, &candidatesCase1, NULL);
1061        sort(oldXORValues);
1062
1063    /* Case 2: XORingDV elements with confirmed XORs to obtain another one. */
1064    printf("\tCASE 2: XORing DV elements with confirmed values... ");
1065    vectorInt32Struct DVElements;
1066    DVElements.length = totalDVhistogram->rows;
1067    DVElements.data = calloc(DVElements.length, sizeof(int32_t));
1068                int index = 0;
1069                for (i = 0; i < DVElements.length; i++){
```

```
                        if (totalDVhistogram->data[i][1] != 0){
                                DVElements.data[index] = i;
                                index++;
                        }
                }

    vectorInt32Struct selectedPrelCandidates;
    selectedPrelCandidates.length = 0;
    selectedPrelCandidates.data = calloc(1, sizeof(int32_t));
    printf(" Searching... ");
    int kdv, kxor;
        for (kdv = 0; kdv < index; kdv++){
                for (kxor = 0; kxor < oldXORValues.length; kxor++){
                        int xored = DVElements.data[kdv] ^ oldXORValues.data[kxor];
            if ((findEqualVector(&oldXORValues, xored) != 0) &&
            (findEqualVector(&oldXORValues, kdv) == 0)) {
               vectorInt32Struct vKdv;
               vKdv.length = 1;
               vKdv.data = calloc(1, sizeof(int32_t));
               vKdv.data[0] = kdv;
               selectedPrelCandidates =
               unionVec(&selectedPrelCandidates, &vKdv, NULL);
            }
        }
    }

    int nThresholdCase2 = excessiveRepetitions(XORDVtotalrepetitions,
    1, totalDVhistogram->rows, "xor", randomnessThreshold);

    vectorInt32Struct candidatesCase2;
    candidatesCase2.length = 0;
    candidatesCase2.data = calloc(selectedPrelCandidates.length, sizeof(int32_t));
    for (i = 0; i < selectedPrelCandidates.length; i++) {
        if (totalDVhistogram->data[prelCandidates.data[i]-1][1] >= nThresholdCase2)
```

```
1104            candidatesCase2.data[candidatesCase2.length] = selectedPrelCandidates.data
1105            candidatesCase2.length++;
1106        }
1107    }
1108    candidatesCase2.data = realloc(candidatesCase2.data,
1109    candidatesCase2.length*sizeof(int32_t));
1110
1111    if (candidatesCase2.length > 0){
1112        printf("%d new value", candidatesCase2.length);
1113        if (candidatesCase2.length != 1)
1114            printf("s");
1115            printf(" obtained. \n");
1116    }else{
1117        printf("No new values. Going on.\n");
1118    }
1119
1120    matrixInt322DStruct newXORDVvalues;
1121    *candidates = unionVec(&candidatesCase1, &candidatesCase2, NULL);
1122    newXORDVvalues.rows = XORextracted_values->rows + candidates->length;
1123    newXORDVvalues.cols = 2;
1124    newXORDVvalues.data = calloc(XORextracted_values->rows, sizeof(int32_t*));
1125    for (i = 0; i < newXORDVvalues.rows; i++) {
1126        newXORDVvalues.data[i] = calloc(2, sizeof(int32_t));
1127    }
1128    index = 0;
1129    for (i = 0; i < XORextracted_values->rows; i++) {
1130        for (j = 0; j < 2; j++) {
1131            newXORDVvalues.data[i][j] = XORextracted_values->data[i][j];
1132        }
1133        index++;
1134    }
1135    for (i = 0; i < candidates->length; i++) {
1136        newXORDVvalues.data[index][0] = candidates->data[i];
1137        newXORDVvalues.data[index][1] = totalDVhistogram->data[candidates->data[i]-1][
```

```
1138            index++;
1139        }
1140        return newXORDVvalues;
1141    }
1142    #endif
```

Bibliografía

[1] D.Falguere and S.petit, “ A Statistical Method to Extract MBU Without Scrambling Information,” IEEE Trans. Nucl. Sci., vol. 54, no. 4, pp. 920-923, Aug. 2007

[2] M. Wirthlin, D. Lee, G. Swift, and H. Quinn, “A Method and Case Study on Identifying Physically Adjacent Multiple-Cell Upsets Using 28-nm, Interleaved and SECDED-Protected Arrays, ” IEEE Trans. Nucl. Sci., vol. 61, no. 6, pp. 3080-3087, Dec 2014.

[3] J.A. Clemente, F.J. Franco, F. Villa, M. bayla, S.Rey, H. Mecha, J.A. Agapito, H. Puchner, G. hubert, and R. Velazco, “ Statistical Anomalies of Bitflips in SRAMs to Discriminate SBUs from MCU” IEEE Trans. Nucl. Sci., vol. 63, no. 4, pp. 2087-2094, Aug. 2016

[4] JuliaLang - Documentación.
https://docs.julialang.org.

[5] C Language - Documentación.
http://devdocs.io/c/.

[6] xCode - Documentación.
https://developer.apple.com/documentation/xcodekit.

[7] Eclipse - Documentación.
https://eclipse.org/.

[8] Qt - Documentación.
https://www.qt.io/es/.